JSC 38605 Rev. A

Guidelines for the Rapid Development of Software Systems

- Revision A -

ENGINEERING DIRECTORATE

AEROSCIENCE AND FLIGHT MECHANICS DIVISION

18 February 1998

National Aeronautics and Space Administration

Lyndon B. Johnson Space Center
Houston, TX

Guidelines for the Rapid Development of Software Systems

- Revision A -

Prepared By:

Denise M. DiFilippo
G. B. Tech, Incorporated

Approved By:

David A. Petri, Deputy Chief	**James P. Ledet**
GN&C Development and Test Branch	**Code Q RTOP Project Manager**
Aeroscience and Flight Mechanics Division	**Aeroscience and Flight Mechanics Division**
NASA/Johnson Space Center	**NASA/Johnson Space Center**

Concurred By:

Karen D. Frank, Chief	**Aldo J. Bordano, Chief**
GN&C Development and Test Branch	**Aeroscience and Flight Mechanics Division**
Aeroscience and Flight Mechanics Division	**NASA/Johnson Space Center**
NASA/Johnson Space Center	

This Page Intentionally Blank

Preface to Revision A

The original release of this document captured the results of work performed in FY'96 with funds provided under the Research and Technology Operation Plan (RTOP) by the Office of Safety and Mission Assurance (OSMA). OSMA has delegated requirements for the Agency Software Program to Ames Research Center Software Technology Division (ARC/IT) located in Fairmont, West Virginia. Work under this initiative was managed at ARC/IT by Kathryn M. Kemp, Deputy Chief, Software Technology Division, and George J. Sabolish, Center Software Initiative Manager. The work was performed in the Aeroscience and Flight Mechanics Division at the Johnson Space Center in collaboration with the Jet Propulsion Laboratory.

The results of FY'96 work were documented in a 2 volume set consisting of:

- JSC 38605 Guidelines for the Rapid Development of Software Systems
- JSC 38606 Guidelines for the Rapid Development of Software Systems - References

This initiative continued in FY'97, with oversight from Center Software Initiative Director William Jackson, with the objectives of determining the effectiveness of the guidelines by using them in a rapid software development demonstration project, comparing the methodology to established standards, and updating these guidelines based on observed results. The results of the demonstration project and one comparison study were documented in the following documents:

- JSC 38607 Correlation of the Rapid Development Methodology to the Software Engineering Institute's Capability Maturity Model
- JSC 38608 Deorbit Flight Software Demonstration Project Summary
- JSC 38609 Deorbit Flight Software Demonstration Lessons Learned

This document has been revised based on observations and lessons learned in the demonstration project. We believe that the result is an improved set of guidelines for the rapid development of software systems.

Though not formally treated here, it is worth noting that some aspects of the Rapid Development methodology have been successfully used to produce actual flight software, developed in the Rapid Development Laboratory, for space craft.

Contributors

The following people have contributed to the ideas and content of this document: Bruce Abramson (Boeing), Aldo Bordano (EG1), Bill Brown(Syscom), Faye Budlong (Draper Labs), Benson Chang (Lockheed), John Craft(EG2), Denise DiFilippo(GBTech), John Ducote (Lockheed), Mark Jackson (Draper Labs), Jim Ledet(EG2), Harry Ohls(JPL), Bill Othon(Lincom), Rose Pajerski (GSFC), Doug Pesek(Boeing), David Petri(EG1), Ron Pierce (Boeing), Carolyn Seaman (Univ. of MD), Sonya Sepabahn(EG1), Nancy Sodano (Draper Labs), Glenn Venables (Boeing), Brad Wissinger (Boeing), Doug Zimpfer(Draper Labs).

Table of Contents

Figures

Tables

Acronyms and Abbreviations

ACRV	Assured Crew Return Vehicle
AFMD	Aeroscience and Flight Mechanics Division
API	Application Programming Interface
COTS	commercial, off-the-shelf
CMM	Capability Maturity Model
DOF	degree-of-freedom
EGI	Embedded GPS/INS
FCOS	Flight Control Operating System
FSSR	Shuttle Flight Subsystem Software Requirements
FY	Fiscal Year
GN&C	Guidance, Navigation & Control
GPS/INS	Global Positioning System/Inertial Navigation System
HIL	hardware-in-the-loop
IV&V	Independent Verification and Validation
ISI	Integrated Systems Inc.
ISO	International Organization for Standardization
ISSA	International Space Station Alpha
IV&V	Independent Verification and Validation
JPL	Jet Propulsion Laboratory
JSC	Johnson Spaceflight Center
MOD	Mission Operations Directorate
NASA	National Aeronautics and Space Administration
OIL	Operator-in-the-loop
OMS	Orbital Maneuvering System
RDL	Rapid Development Laboratory
RTOP	Research and Technology Objectives and Plan
SEI	Software Engineering Institute
SLOC	Source Lines of Code

1.0 Overview

The Aeroscience and Flight Mechanics Division (AFMD) at the National Aeronautics and Space Administration-Johnson Space Center (NASA-JSC) Engineering Directorate is exploring ways of producing Guidance, Navigation & Control (GN&C) systems more efficiently and effectively. A significant portion of this effort is software development, integration, testing and verification.

To achieve these goals, the AFMD established the GN&C Rapid Development Laboratory (RDL), a hardware/software facility designed to take a GN&C design project from initial inception through hardware-in-the-loop (HIL) testing and perform final GN&C system verification. The operations approach for the RDL concentrated on the use of commercial, off-the-shelf (COTS) hardware and software products to design and develop the GN&C system in the form of graphical data flow diagrams, to automatically generate source code from these diagrams and to run in a real-time, HIL environment under a Rapid Development paradigm.

The success of these efforts motivated further study and documentation of Rapid Development methodologies. The initial goal was to formalize the successful methods used to date in the GN&C RDL. Subsequently the team expanded on these methods, based on knowledge gained from extensive search and study of the current literature. The resulting methodology was documented as a guidebook for Rapid Development.

The methodology was tested and observed by applying it to a significant demonstration project. Members of the project team reviewed the effectiveness of the methodology on an ongoing basis. Improvements and corrections to the methodology and the RDL facilities and infrastructure evolved along with the demonstration project. At the conclusion of the project, the team met several times to reflect on the project, methodology, and lessons learned. These reflections have been documented as JSC 38608, *Deorbit Flight Software Demonstration Project Summary* and JSC 38609, *Deorbit Flight Software Demonstration Lessons Learned.*

Based on the observations and lessons learned during the demonstration project, we have refined the rapid development methodology, as presented here. This is in step with the overall philosophy of Rapid Development: to revise the plan, based on lessons learned, before moving on, thus reducing risk by finding problems early in the development cycle. In the spirit of continuous improvement, the team welcomes comments and feedback, especially any observations from those who have practical experience using this or a similar methodology.

2.0 Introduction

What is the best way to develop systems that include software as a significant component? For many years, the “gold standard” of software development has been the use of structured analysis and programming in the context of the “waterfall” model of a system life-cycle (see Figure 1. The Traditional Waterfall Flight Software Development Approach on page 3). In recent years, a modification to the waterfall model (Incremental Development) which partitions large systems into independent deliverables and then sequentially applies the waterfall model to each subset, has gained popularity.

2.1 Motivation for a new methodology

In the context of many of today’s systems problems, the waterfall model approach to system development, and its modified incremental development approach, are often ineffective for a variety of reasons.

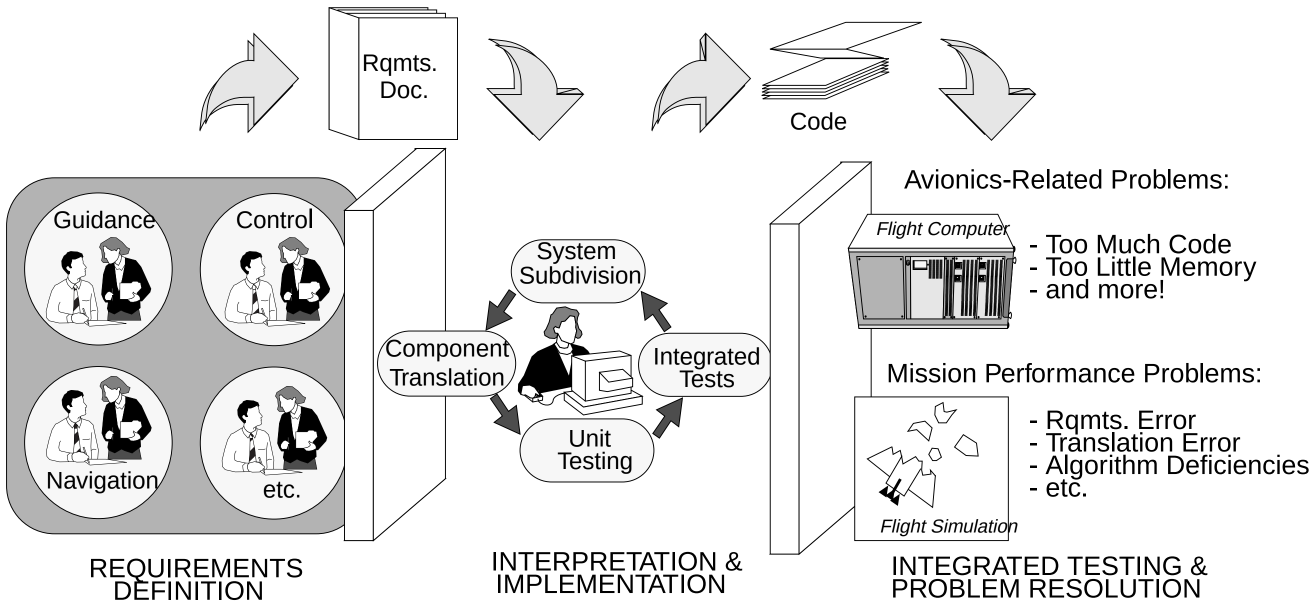
As system complexity increases, it becomes more difficult to completely specify detailed requirements in text form. The documents that attempt to describe these systems become large and complex. The requirements may interact in intricate and complex ways. The review and sign-off processes can be lengthy and expensive. Verifying that the requirements documentation is complete, accurate and consistent can be a daunting or impossible task.

As the problems to be addressed increase in complexity, the solution approaches become less obvious. It may not be reasonable to ask a user community to enumerate requirements, since technology may be able to offer approaches never before used. That is, we have gone beyond using software to just duplicate human effort faster. Still, software developers usually will not be experts in the domain of the problems to be solved, so it is similarly unrealistic to depend solely on them to define a system. A cooperative effort, among domain experts and technology experts, to discover system requirements can leverage the value added of new systems. The waterfall methodology often does not accommodate this philosophy, since requirements are developed independently and “thrown over the wall” to implementors who may have no knowledge of the system beyond that written in the requirements documents.

The pace of change coupled with the potentially long lead time to develop systems often creates the dilemma of today’s new systems meeting yesterday’s requirements. This is especially true when system requirements are completely and contractually specified and fixed early in the development cycle. If the requirements are handed off to implementors who are completely separate from the domain experts, then implementors are likely to be unaware of important changes that occur during the development cycle. This effect is compounded when the problem domain exists in an area where the state-of-the-art is changing at a rapid pace.

In the GN&C domain, software development includes some additional challenges. Avionics hardware often is being designed simultaneously with the software, so some requirements and performance criteria may not be known during the early stages of software development. Specialized operating environments and processor platforms impose challenges to creating effective test environments. Often specialized languages and operating systems are dictated by the real-time performance requirements, which may imply separate development and run-time environments. In such an environment, traditional test strategies, which perform integration and testing after coding is completed and often depend on “testing in” system quality, are often ineffective and generally quite expensive.

Ultimately, the need for a better way to develop software systems is driven by the need to manage the risks involved. These include development costs, maintenance costs, and the more difficult to measure cost to an organization when it does not have the best system for the customers’ needs. The bottom line is that we need better, faster, cheaper software systems.



Engineers skilled in particular problem domains formulated detailed requirements for the systems & subsystems.

Other organizations interpreted the requirements and translated them into computer code.

Unforeseen problems arose deep into the schedule during integrated testing & simulation.

Figure 1. The Traditional Waterfall Flight Software Development Approach

2.2 Finding a better methodology for modern software development

While many system development efforts still claim to use the waterfall model, in the trenches programmers, analysts and project managers are devising more effective techniques. Today, these have to be forced into the waterfall life-cycle for external consumption; that is, to pass reviews, quality gates and sign-offs.

Is there a way to capture these more effective techniques and mold them into a life-cycle model that is effective in today’s software engineering environment? Suggestions for doing just that are presented in this guidebook.

The project team incorporated practical experiences gained using techniques that facilitate the Rapid Development of high quality systems, especially in the context of GN&C flight systems. Using what we have learned, over time and with testing and validation, guidelines for Rapid Development and a system life-cycle model for Rapid Development were constructed.

This document encompasses several major topics relevant to Rapid Development of quality software systems. These include:

- Important issues, concepts and practical ideas that, based on the experience of the RDL staff and extensive research of the software community, support success in the Rapid Development of high quality systems, especially for GN&C applications.
- The phases of a proposed new life-cycle model, including the major topics of interest in each phase. This model is intended to be a formal systems engineering approach to modern software development.
- Project management issues using a Rapid Development approach. Classical development theory is rich with suggestions for managing the cost, quality, schedules and risk associated with software development. In adopting a new life-cycle, these issues must also be addressed.
- Processes that support software development and project management in a Rapid Development environment.
- Collecting and evaluating metrics in a Rapid Development environment. After studying the current state of the art of metrics data collection and evaluation, recommendations are presented for a start up metrics program in a Rapid Development environment. These include essential modifications to standard metrics so that they better support the new methodology.
- Lexicons of important Rapid Development and metrics terminology.
- References.

2.3 The Expected Payoff

Experience with projects that used some of the techniques of the Rapid Development methodology, indicates that high quality systems can be developed faster and with smaller, but more integrated teams, using this technique as opposed to the waterfall approach. Furthermore, experience shows that user satisfaction with the systems developed improves when using Rapid Development techniques. By defining and then applying a rigorous model that can consistently and dependably produce these results, it is anticipated that high quality systems can be developed with less risk, lower cost, and better adherence to schedules.

Some reasons for the observed success of this methodology include:

- Complete integrated systems are built early in the development cycle.
- Early integrated systems are often low in fidelity, with stubs for unavailable software or hardware components.
- “Systems” problems and interface problems are solved early.
- Due to the concurrent engineering approach, staffing requirements tend to be relatively level for much to the development effort.
- Prototype software and hardware systems are not thrown away, they evolve into the

final product.
- Early integrated test builds customer and developer confidence.
- Milestones are determined with specific product focused objectives and acceptance criteria.
- Detailed development is done by Domain Experts.
- Integrated Rapid Development project teams are formed around the skills and expertise required to complete the project, including domain expertise, systems expertise, and technical management
- The project team is responsible for:
    - integrating all project elements, and
    - Configuration Management and Quality Control, and
    - ensuring the project remains product focused.
- The project team takes ownership of the entire development process and end product.
    - End-to-end responsibility and ownership is more efficient and promotes a more productive work environment.

Based on our knowledge and experience with Rapid Development, we also anticipate that large on-going projects, and especially in the real-time GN&C domain, will benefit from:
- The use of COTS standards, languages, development environments, and test tools
- The use of commercial standards, processors, operating systems, and data bus architectures
- Test facilities that more closely duplicate the operational system, as a result of these first two
- Increased flexibility to use the developed systems in multiple facilities, for example training and vehicle check out
- Increased capability for reuse, growing out of the use of COTS tools and industry standards across multiple projects

One important aspect of Rapid Development, the Spiral Development Process is an accelerated development process where the system requirements, design, code, test, and integrated test processes are iterated on concurrently rather than being executed sequentially and in a disjointed fashion. A spiral development process can make effective use of tools to integrate the requirements analysis, design, code and test (including test coverage) environments.
- An integrated environment involves the designer in all phases of product development.
- Changes anywhere in the requirements, design, or code are more easily implemented everywhere.
- CASE tools, including graphical user interface (GUI) simulation and modeling tools, coupled with autocode generation and real-time processor testing, are most effective in the hands of the rapid development team and speed the spiral development process.

The development tools available today and anticipated for the near future, such as CASE GUI simulation and modeling tools, are *not* just more advanced programming languages.

- Tools support integrated analysis, design coding and testing efforts.
- If an integrated toolset is not used in the requirements and design phase, then an additional step to translate logical flow into data-flow code is required.

3.0 Guidelines for Rapid Development

These are some of the most important features of the Rapid Development methodology that is successfully evolving in the GN&C RDL and as a result of this research:

3.1 Project Staff

Creation of a small core team that is talented, knowledgeable in all key areas of the project, and able to follow the project for its entire life-cycle is critical. Most of the team members should be dedicated to the project full time especially after the initial concept development phase is completed. Rapid Development methods depend on rapid situation evaluation and response, and leveraging knowledge from one phase to the next. Construct the team carefully, and try to keep them together. If possible, once a team establishes a strong working relationship it should be maintained even across projects (although modified as needed to achieve the critical mix of expertise for each project). A working Rapid Development team is a valuable asset.

Involve the users and customers. Get commitments from them to be involved in solving the problems, reviewing the system, acceptance testing, etc. Make sure the project plan emphasizes the importance of this involvement. Part of what makes this approach rapid is getting things right as soon as possible to minimize misinterpretation, redesign, rewrites and change requests to already completed work. A close working relationship among the project team members and the users and customers helps make sure that the delivered system is the desired system

Users/Customers may exhibit marked differences in preferences and satisfaction levels. It is generally a mistake to assume that the developers will be able to negotiate a universally acceptable solution. Assign someone to have authority and responsibility for resolving conflicting requirements and desires.

Because the project team members are all intimately involved in the design and development, they may not maintain the objectivity necessary for full, independent, validation and verification. Plan to recruit knowledgeable sources or hold a periodic independent review outside the project team to assist with these efforts.

The powerful capabilities of some modern development environments allow domain experts, who are not necessarily software experts, to design, implement and test complex software. It is sometimes tempting to ignore or under-represent software systems specialist expertise in a Rapid Development Team. In general, this software engineering expertise remains important and should not be neglected in the team makeup, especially with respect to architecture, integration, and performance aspects of a system. This emphasis is especially critical when dealing with real-time, embedded, safety critical systems. Similarly, the capability to perform traditional project control functions, such as configuration management and test planning, should be represented on the team.

3.2 Tools to Support Rapid Development

The use of advanced software engineering tools can greatly improve productivity. Graphical development tools, especially when combined with auto-coders, provide a powerful development environment which can be used by domain experts as well as software professionals. Integrated simulation capability can put a powerful, accessible, realistic, and easy to use testing environment in the hands of implementors. Investing in appropriate tools and standards can add to the flexibility and robustness of the system, and potentially improve the ability of the team to respond to problems and changes, by supporting multiple operational and test configurations.

Automated support for design, coding, testing, documentation, and configuration management are among the desirable options. The use of an automatic code generator, especially, mitigates the need to track and solve many problem areas, such as syntax errors, typographical errors, and some common programming errors. (However, the quality of auto-generated code needs special attention, due to the current state-of-the-art of this type of tool. This seems to be improving as the industry matures and more vendors are entering the market.)

Historically, real time systems often depended on “home-built” custom languages, operating systems, development and test environments, data buses, and even processor platforms. These became necessary when COTS products were not available to support the application requirements for speed, scheduling precision, size of executable code, process synchronization, fault tolerance or hardware robustness. Today, more and more often COTS products are available to support all aspects of these systems. Now, it is generally quite satisfactory and overall cheaper to use these products rather than design, implement, maintain and support custom development of the hardware and system software, including the development and test environment.

Advanced quality assessment tools can also be quite useful. This is an area that is coming of age, and major improvements are anticipated in the near future. When added to the developers’ toolset, these tools perform SR&QA types of analysis early in development, to support efforts to build quality into systems from the earliest releases.

Similarly, advanced tools to support configuration management, documentation control, discrepancy reports, change requests, and other support process areas, can be used to improve both inter-project communication and productivity. Even simple measures such as making all project documentation available on web pages will eliminate sources of error and prove useful to improving developer productivity, since all team members will have access to all current documentation. Additional tools which may, for example, correlate test results to discrepancy reports and release versions can also be useful and are becoming available for an increasing number of platforms and development environments.

Test plans, test results, discrepancy reports and resulting modifications can all be stored electronically and be accessible across the network. Using currently available tools and web-based technology, it is possible to link the information in the documents, showing, for example, which requirements are tested by each test case and even linking to the test results. The electronic access and linking are powerful ways to improve access to information and to eliminate errors of transcription and oversight. In addition, the links can give important insights into progress and status.

Plan for element reuse. Organize code to facilitate reuse. Invest in hardware, software and staff to create and maintain a reuse library. The library should support browsing capability, with searchable attributes, and include tailoring instructions for library elements. Investment upfront in reuse pays off long term. There are costs associated with establishing, populating and maintaining a reuse library. Management and budget support is crucial.

Implement automated release build capability to consistently interact with configuration management and build releases from known libraries.

Tools to support Rapid Development are currently proliferating rapidly, and capabilities increase frequently. Plan to invest time investigating and evaluating new tools on a regular basis. As useful and appropriate tools become available or improve, add the new capability to the toolset. Do not assume that the tool used for one project is best for the next, or try to force fit a tool where it does not belong just because the team is already familiar with it or the lab already owns it. Choose the best tools for any particular task. Fast improvements in the quality and capability of these tools mean that it can be important and effective to invest the effort required to stay on the leading edge of the Rapid Development technology.

3.3 Approaching the Problem

Work the system architecture, including interface requirements, inter-project dependencies, test requirements and validation strategies, very early in the project life-cycle. Under classical development approaches, integration, communication, and interface issues have frequently been major problems. Under Rapid Development, it is recommended that end-to-end integration of the architecture begin early in the development cycle, so that many of these types of problems are resolved before the rest of the system has been developed, or addressed throughout the development. Dummy software stubs are usually sufficient to test the software and hardware connectivity in the early phases. Simulations, using mathematical models, are typically substituted for planned hardware modules. Retest the architecture as actual hardware and software become available. It is crucial to identify and validate the system architecture as early as possible.

Once the system architecture is in place and functioning, and before proceeding with more detailed implementation, consider addressing the issue of useful system utility functions and any special project development standards. Create the project utility library from existing libraries if possible, or at least use the current project to begin or augment the reuse library. Perform code inspections on the utility functions; these are important building blocks. The team must agree on and be familiar with their function. And, building and inspecting these utilities is a good way to solidify project coding and development standards.

Early documentation should emphasize functional requirements rather than design and implementation issues. Design documentation should emphasize system architecture and interface requirements. Detail level for specifications should vary depending on risk level of the system element; defer detailed elaboration of the low risk elements until the high-risk elements of the design have stabilized.

Start tackling the hard problems first. These are high risk areas for a project. Use prototyping to test alternatives and choose the best (or reduce the set by eliminating alternatives found to be unworkable). As a project progresses, risk and uncertainty will be reduced by this approach. Some of these prototypes may be throw-aways. Once simple prototypes have helped find the correct way to solve a particular problem, go back and implement or evolve a production version (usually more robust and with more error checking and recovery modes than the prototype).

Perform hardware-in-the-loop testing earlier in the development cycle than has been historically common. As soon as actual flight or support hardware is available, and testing with it is practical, integrate it into the system, at least for testing. Again, the emphasis is on identifying potential problems as soon as possible and fixing them while it is still relatively easy to do so (i.e., without major reworks).

As much as possible, “as-built” documentation should be automatically generated by the software engineering tools. Documents should evolve with the system; add information to the appropriate documents when it is known and stable, delaying complete formal documentation of specific issues until after problems have been surfaced and successfully resolved.

3.4 Implementation Hints

Careful planning and a firm resolve are required to guard against “requirements creep” under a Rapid Development methodology model. This is the tendency to continue to demand additional functionality from a system until it has crept beyond the original scope of the project. This is a special risk in this type of development since one of the key characteristics of Rapid Development is the evolution of (detailed) requirements during the development phases. Clear traceability of requirements to distinguish “derived” requirements from “new” requirements is necessary. Set goals for each evolution cycle and for the overall project that will clearly identify when the project is finished. Consider the cost/benefit trade-offs whenever plans are modified.

Create and maintain automated test sequences for each system developed. Augment and rerun the test series to validate that system changes have not interfered adversely with existing capability. This applies to both development and maintenance cycles.

Plan for project turnover, from development to sustaining engineering, in parallel with design and development by involving the users and transition team. This can begin as soon as the design stabilizes.

Rapid Development does not imply ad hoc development. It is a fast paced, dynamic environment, typically with tight schedules and high expectations. Careful planning and monitoring is essential for success.

Many of the observations regarding the effectiveness of Rapid Development techniques have been learned and improved while working on small to medium sized projects. The concepts need additional testing on large projects, since issues of scale may well demand modifications to the techniques.

4.0 A Proposed Life-Cycle Model

In the long run, any systems development effort will have many common tasks, no matter what methodology is used. That is, requirements must be determined, code must be written, tested and validated, documentation must be written, and the project must be managed. There are various ways to order and perform these functions. The life-cycle model proposed in this document uses as its basis the model which has been successfully evolving in the GN&C RDL and augments it in ways that are designed to improve project management, software control, and verification and validation.

The proposed life-cycle model, from idea to obsolescence, including both system development and system maintenance, is comprised of the following major phases (also see Figure 2. Life-Cycle Major Phases on page 11):

•Project Initiation

•Project Evaluation

•Conceptualization

•Evolution

•Finalization

•Installation

•Sustaining Engineering

•Shutdown

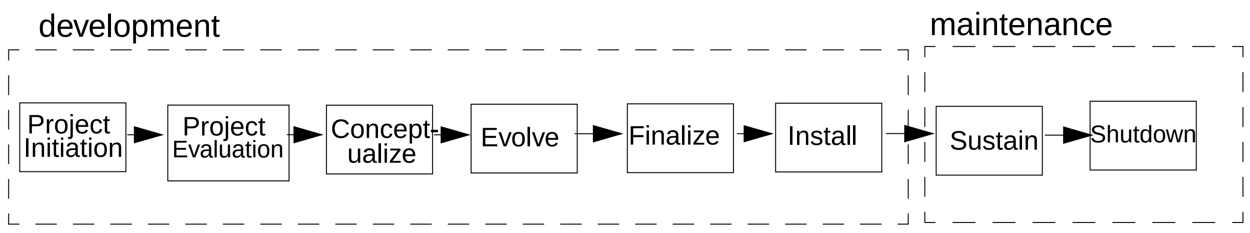


Figure 2. Life-Cycle Major Phases

The high level objectives of each phase are discussed below. A later section will address support processes for the model.

This document primarily addresses the development phases of the project life-cycle model. Under the Rapid Development paradigm, the guiding concept is “build a little, test a little, fly a little”. This approach tends to focus on design problems, technical issues, and implementation errors early in the development, before they propagate and while they are easier and cheaper to fix (relative to modifications made closer to or after delivery of a completed system). Using this approach, it is critical to maintain interaction with the target community (users and customers) throughout the development cycle. Similarly, documentation, project plans, schedules, and software releases are living entities under Rapid Development, to be revised and augmented as a project progresses, as more is learned about the problem to be solved, and as more details evolve and are implemented and validated. These critical interfaces are illustrated in Figure 3. Rapid Development Team Critical Interfaces on page 12.

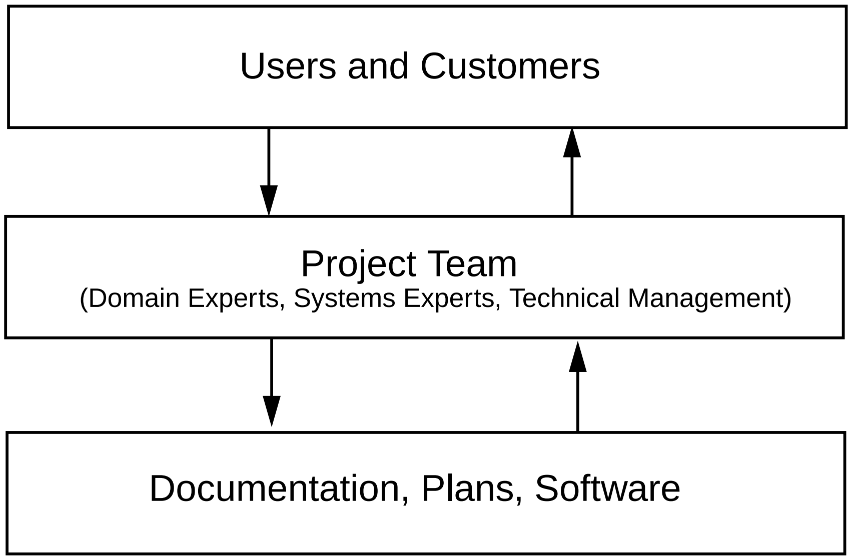

Users and Customers
Project Team
(Domain Experts, Systems Experts, Technical Management)
Documentation, Plans, Software

Figure 3. Rapid Development Team Critical Interfaces

5.0 The Development Phases of the Life-Cycle Model

5.1 High Level Objectives of Project Initiation Phase

The Project Initiation phase is the first step in determining new and potentially promising projects deserving of further study. The overall goal is to determine whether a project is needed, feasible (both in technical and budgetary terms), and compatible with the goals of the organization. Typically this phase will be user/customer initiated.

The purpose of this phase is to collect and present sufficient information about the problem and the proposed system, or system upgrade, to support a management decision about whether to proceed to the next project phase. This will include information about the skills required to perform the next phase.

5.1.1 The Proposal

The Initiation phase begins when someone identifies a problem and proposes to develop or upgrade a system to help solve it. The first step is to identify and state the problem to be solved.

Determine the high level functional requirements which must be met in order to solve the problem. In this phase, there is no need to define how the problem should be solved. Requirements need not be complete or detailed. Any hard requirements must be stated. Document what is known about the problem. Have customers complained? Are deadlines being missed? Is there a budget problem? Have changes occurred which require support in an entirely new area? In general, why must this problem be addressed?

State the known issues and identify areas of uncertainty. If there are known risks, identify them along with any assessments of the risk level and suggestions for managing the risk.

What will determine whether the problem has been solved? In general terms, state the success criteria from a business standpoint as well as technical and performance criteria. Include any known coordination issues with other projects or deadlines, as well as any known interface requirements.

Initial feasibility analysis is appropriate, but extensive analysis is not necessary at this phase. Just state what is known or an informed belief about the feasibility of the proposed project. Suggest alternative solution strategies. Identify the known relationships between the proposed project and other projects, including existing systems, systems under development, and others being proposed.

5.1.2 The Decision to Proceed

If the proposal is accepted, a decision is made to proceed to the Project Evaluation phase. This is not a commitment to the full project life-cycle. It is a commitment to further investigate the problem and the proposal.

To close out this phase, management should commit the necessary resources to the next phase. These include personnel, budget, equipment and training.

Personnel include a project advocate, or leader, who leads a rapid development team of systems experts who have or will be trained in the necessary skills and domain experts with expert understanding of the problem being addressed.

The next phase is not intended to be a full detailed requirements analysis. Management should clearly outline the budget, equipment and time frame which are being committed to the evaluation phase. Initial estimates of the overall project costs should also be made at this time. These will undergo refinement in succeeding project phases, if authorized.

It may also be appropriate to identify and pursue training and purchasing requirements, especially if they are needed for the next phase or if long lead times are involved and the project is likely to be funded. Are the training and purchase requirements consistent with maintaining state of the art capability in Rapid Development? If so, then making the investment now has the double payoff of improving overall capability while contributing to the momentum of the current project.

5.2 High Level Objectives of Project Evaluation

Here is where the project team really begins to look at how to solve the problem at hand. Working with the domain experts, the team will study in detail the problem to be solved. Rather than assign individuals to write various parts of the requirements, the core team should initially meet as a group to exchange knowledge about the problem, discuss solution strategies, and consider options. The goals in this phase are to understand the problem and devise a solution strategy.

5.2.1 The Evaluation

The high level functional requirements will be produced in this phase, but they may look a bit different from traditional requirements. There may be areas in which two or more alternative sets of requirements are identified as possible solutions, or the requirement may be stated as a range. Some requirements may be desirable, others may be strictly required. Some areas may be unknown.

Part of this effort is feasibility analysis. The team should suggest and evaluate alternative solution strategies. These evaluations should consider (at least) cost, risk and probable outcome. Are there interdependencies with other projects, and if so how will that effect the feasibility of this effort? This may include technical dependency issues, but may also involve schedule dependencies and staff and resource sharing. Can these dependencies be exploited in ways that benefit this and other projects? At the conclusion of this phase, most alternatives should have been eliminated. It is, however, perfectly acceptable to enter into the implementation phases with some of these decisions still unmade, so long as the project plans include a strategy for further evaluation and decision making. The evaluation may include, for example, prototyping parts of the system in a variety of ways and applying selection criteria and tests to aid in decision making.

Another part of this phase is risk assessment. Identify the potential show stoppers, the really difficult parts of the problem. The difficulties may be technical, budgetary, time related, complexity related, or uncertainty about the problem or domain. Assess the level of risk. Compare various possible solutions. What is the impact of not doing this project, including risk to other projects or programs? Cost/benefit analysis may be useful. Plans for controlling risk should include early attention to the high risk areas, with reviews and decision points built into the schedule to reevaluate the risk potential. This will tend to contain the risks, by delaying implementation of the majority of the system until after the most challenging areas have been successfully designed. Similarly, in the project plans, detailed design of the more straight forward parts of the system should be delayed until after the design for the high risk areas has stabilized. The well defined, low risk, parts of the project should be relatively easy to design around the tough parts after they have been solved. The reverse strategy can be quite costly, and often involves extensive redesign and rewrites and can result in code that is more difficult to maintain.

Other issues that the project team should address in this phase include safety, security and privacy concerns, and initial estimates of the types and quantities of resources (staff, hardware, commercial software, other) required to implement the system.

When all of these issues are well understood, the team should choose a development strategy for the project. While this document is emphasizing a new Rapid Development methodology, each project should carefully consider whether this or another strategy is more appropriate for the problem being addressed.

For large projects, the team may wish to subdivide the effort into smaller efforts. In this case, the project partitioning should be defined, and a work breakdown structure should be defined. Project teams must be identified for each partition, and each team should proceed with development, beginning with this (Project Evaluation) phase. Plans for coordination and integration of the partitioned efforts must be specified.

5.2.2 Choosing a Development Strategy

Many of the ideas for improved software development methodology can be applied to any project. Yet each problem is different, and the project team needs to decide what is the best methodology to solve a particular problem. Tailor the guidelines for maximum success, and document the process to be used for each project as part of the project plan.

We distinguish among three primary approaches to software implementation: Waterfall, Incremental, and Evolutionary.

5.2.2.1 The Waterfall Model

The traditional Waterfall approach is characterized by distinct, sequential development phases, with separate hardware and software development paths and no integration until late in the life cycle. The exact number of phases and their definition may vary somewhat depending on project size and organization culture, but typically include the following:

- Requirements Definition
- Requirements Translation (System Design) and Review
- Software Design and Coding

- Software Test
- Integrated Test

The phases are often organized into “silos”, or distinct organizations often located far apart. Each technology discipline takes ownership of only a portion of the final product, and for only a certain phase of the program. Major milestones are well defined, and under government standards typically include:

- SRR: System Requirements Review:
- SDR: System Design Review
- SSR: Software Specification Review
- PDR: Preliminary Design Review
- CDR: Critical Design Review
- TRR: Test Readiness Review
- PCA, FCA: Physical and Functional Configuration Audits
- FQR: Formal Qualification Review

The classic Waterfall development model is best used when system requirements are straightforward, well understood and stable. The problem to be solved should be one that is well understood, with standard solutions. Funding should be stable and predictable.

5.2.2.2 The Incremental Model

The Incremental Model is characterized by a series of waterfall cycles that together complete a project. Usually only 2 or 3 cycles will be used for medium sized projects, often as many as 5 to 10 cycles for large complex systems. A system is delivered after each cycle with some subset of the final desired functionality but with each delivered function complete. For example, a timekeeping system might deliver the capability to support weekly operations in cycle one, periodic reporting functions in cycle two, and planning and forecasting functions in cycle three. This approach partitions the total problem to deliver some useful capability earlier than it would be possible to deliver the entire system. The incremental model is sometimes (mistakenly) used to claim that “Rapid Development” is being used within the standard government milestones.

An Incremental development strategy is recommended when the most critical functions required of the system are well understood and the project is not small. The system must lend itself to being divided into separate, complete, useful, stand-alone subsystems. These subsystems will usually be of varying levels of criticality, with the more critical functions implemented in earlier deliveries. The incremental approach helps ensure that highly critical functionality gets delivered as soon as possible, which is especially important under uncertain funding conditions. Pre-planned product improvement cycles are based on the Incremental Model.

5.2.2.3 The Evolutionary Model

In the Evolutionary Model, an integrated system is developed early and incrementally improved toward the final goal. The driving philosophy is “Build a little, Test a little, Fly a little.”

The evolutionary model resembles, but is not the same as, the original Spiral Development model as presented in the literature (especially Boehm, 1988). The original Spiral model is a type of evolutionary model in which the spirals are driven by a philosophy of risk reduction. Another similar approach presented in the literature relies primarily on rapid prototyping to discover system requirements before proceeding with implementation.

The Evolutionary Model builds on these approaches. A cyclic process is used to rapidly execute a development cycle. All activities (detailed requirements discovery, design, coding, testing) are essentially performed concurrently within a cycle. Requirements tend to evolve with the design. Each cycle has specific goals. The goals may be chosen, for example, to contain risk, achieve a desired level of fidelity, implement specific functions, or coordinate with other project schedules. The results of each cycle help determine the goals for succeeding cycles. Each cycle results in a complete, end-to-end, system. Hardware-in-the-loop testing is initiated earlier in the development cycle than with traditional methods. A rapid development team includes all the necessary skills and expertise and takes ownership of the entire process and end product.

In an Evolutionary Development process, the level of fidelity of the project increases over time until it is completed. The project is split into a series of logical milestones, or “Drops”. Each Drop represents an increased level of fidelity.

A Evolutionary Spiral Development methodology is recommended when:

- System requirements are vague or incomplete.
- The problem to be solved is new or not well understood. Solutions unknown, uncertain, or not obvious.
- Software development must occur concurrently with hardware development, contributing to the risk and uncertainty.

5.2.3 Rapid Prototyping

Any of these three development strategies, but especially the Evolutionary Development model, may be enhanced by the use of rapid prototyping techniques. That is, for those aspects of the system for which the best solution is not known, “quick and dirty” prototypes of alternatives can be employed to aid in decision making and drive the final design. These are typically not robust enough to be delivered, and are used to illuminate problems and alternatives and solidify design and implementation approach.

In many cases, these prototypes can be evolved to become the delivered code. In some cases, the prototypes are too rough, and it is more efficient to rewrite the prototype module as a robust system component. In either case, by using the same person or team to do both prototyping and final development, all of the knowledge gained in the prototyping effort is used in final implementation, and this continuity tends to improve both the speed and quality of implementation.

5.2.4 Hybrid Approaches

In many cases, some hybrid approach may be preferred. This is especially true for relatively large projects where the various project partitions may be developed using different approaches, or the high level project management may proceed differently from detailed sub-system project management.

5.2.5 The GN&C RDL Preferred Model

In the GN&C RDL, an Evolutionary Spiral Development methodology enhanced by Rapid Prototyping techniques has been successfully applied. This approach will be assumed for the remainder of this document (see Figure 4. Evolutionary Development Life-Cycle Model on page 18).

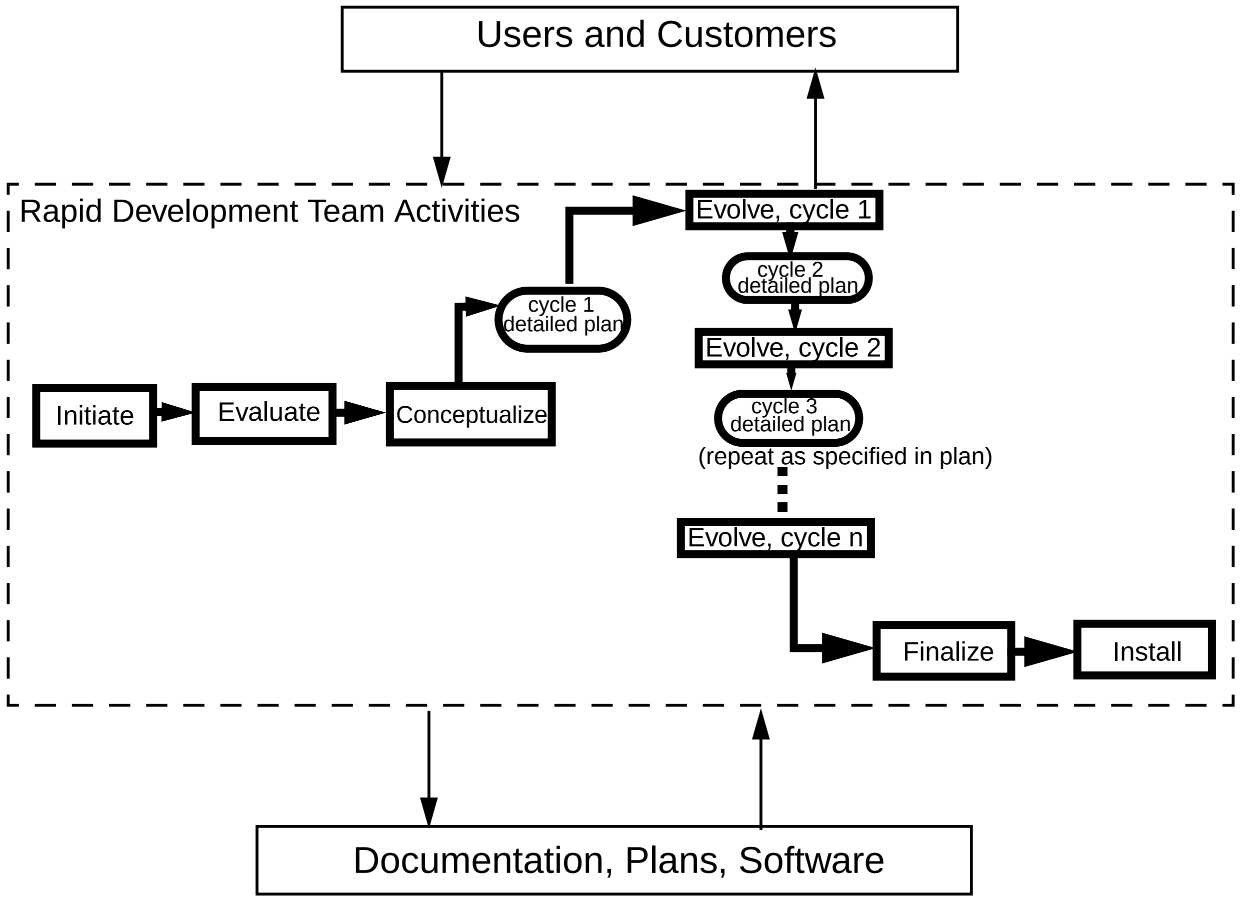


Figure 4. Evolutionary Development Life-Cycle Model

5.3 High Level Objectives of Conceptualization Using an Evolutionary Spiral Development Life-Cycle

Entering this phase, the project team has a good understanding of the problem to be solved, the risks involved, possible solution strategies, and has chosen a development approach.

In the conceptualization phase, the team prepares for implementation. The software engineering environment that will be used for implementation, including tools, facilities, hardware, processes and procedures, should be defined, procured if necessary, and installed.

The primary deliverables of this phase are the system level functional requirements, the high level system architecture, and the project implementation plans. If applicable, an analysis of dependencies of this project on other projects and systems may also be produced in this phase.

5.3.1 The Functional Requirements

The functional requirements document should be fairly detailed as it will drive the system implementation in later phases. Note that no detailed requirements or detailed design document is completed prior to implementation. In our evolutionary Rapid Development environment, implementation and design details will be discovered as the system is implemented. It is therefore imperative that the functional requirements document be a clear, solid and complete description of the required functionality of the system.

5.3.2 The System Architecture

High level system architecture may be specified by diagrams, but whenever possible it is also desirable to implement a working prototype of the architecture design. This can be an extremely low fidelity implementation, with most functional modules stubbed out, if necessary. By implementing an integrated end-to-end prototype early in development of the system, many interface issues can be solved before the system has been coded. Experience has shown that these problems are easier and cheaper to fix early in the implementation of a system, when less code will have to be rewritten to accommodate new interfaces. To the extent possible, it is desirable to perform hardware-in-the-loop testing on this early system prototype. Here again, the goal is to identify and fix potential interface problems early in the development of the system.

After any problems that surfaced have been solved, the resulting prototype (often called the Phase Zero implementation) serves as the initial high level system design. Note that the design is implicit in the successful implementation, rather than design driving the implementation. This gives implementors flexibility to evolve the best design that both works and fits the functional requirements. “As-built” design documents should be produced, and should evolve with the system’s implementation, but it should be possible to automate much of the work to prepare such documentation.

5.3.3 System Dependency Analysis

In some cases, a system dependency analysis may be useful.

The dependency analysis should include functional and data dependencies between this and other systems, existing or planned. Any assumptions made for this system that imply levying requirements on other systems should be called out, and such information should be communicated to the appropriate project teams.

Dependency analysis should also include reusability analysis. Look in the reuse library for existing code that can be used on the current project, or can be easily modified for use on the current project. Also, identify areas where work done for this project could benefit other projects.

5.3.4 The Project Implementation Plans

Two levels of project implementation plans should be produced. The overall implementation plan will show plans for the remainder of the development phases of the system life-cycle at a moderate level of detail. A more detailed plan will be prepared for the cycle which immediately follows this one, Cycle One of System Evolution.

It may be tempting to achieve verbal, ad hoc, team consensus on release plans, especially detailed plans for a single cycle, and “save time” by avoiding the formal effort to write the plan. While these plans need not be excessively detailed documents, it is important to capture the major points in writing and that the plan be accessible to and used by the team to drive schedules and priorities of individual team members, maintain coordination among the team, and encourage conformance to plans and schedules.

5.3.4.1 Implementation Plan for remainder of project development

The overall plan for implementation of the system should include:

- Goals
- Scope of Effort
- User/Customer responsibilities
- Deliverables
- Number of Evolution cycles (Build strategy)
- System level objectives of project and of each cycle
- Preliminary Project Schedule
- Cost Estimates
- Procurement plan
- Verification/validation requirements
  - (including Test approach, strategy, and requirements)
- Configuration management plans
- Documentation requirements

The implementation plan is not a static document. As implementation progresses, it should be updated (at least at the end of each evolutionary cycle) to reflect current knowledge of the project. In general, certainty should increase with each cycle and plan update.

5.3.4.2 Detailed Plan for cycle one of System Evolution

Initially, the detailed plan for evolution cycle one should be prepared. Then, for each evolution cycle, one of the exit conditions of the cycle is the completion of a detailed plan for the next cycle. Lessons learned in the cycle and implications of the detailed plan for the next cycle may impact the overall implementation plan, in which case it should be updated as well.

Topics which should be considered, and included as relevant, for the cycle detail plan include:

- objectives

- constraints
- alternatives
- risk areas
- schedule
- cost estimates
- planned deliveries, including documentation and updates
- Verification/validation plans
    - test strategy, including HIL, OIL, and end-to-end integration tests
- configuration management issues

5.4 High Level Objectives of System Evolution Phase

This is the phase where the majority of system implementation takes place. The primary goal is to evolve the system according to Overall Plan developed in the Conceptualization Phase.

The evolutionary phase will typically be divided into several sub-phases, called “cycles”. The number of cycles planned and the major system level objectives for each cycle will have been defined during the Conceptualization phase. As a rule of thumb, objectives should get more concrete with each succeeding cycle.

Typically, the plan for a cycle will call for maturing some subset of the system functions to specified levels. It is often a good development strategy to concentrate on the more difficult, less understood, more risky modules first, as prototypes. This way, each cycle reduces uncertainty in the project and its budget and schedule.

Each cycle should include user/customer evaluation and documented feedback. The following cycle should address this feedback. This increases the likelihood of achieving a high level of user and customer satisfaction with the final product.

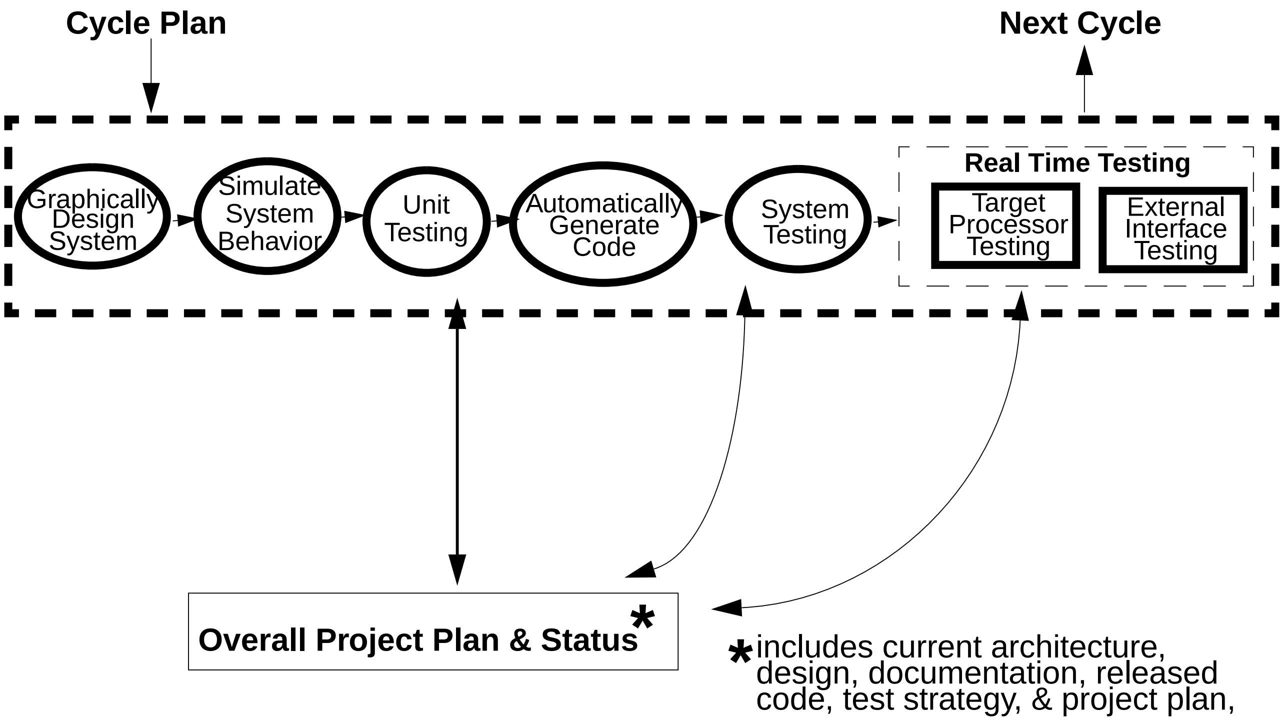

Cycle Plan
Next Cycle
Graphically Design System
Simulate System Behavior
Unit Testing
Automatically Generate Code
System Testing
Real Time Testing
Target Processor Testing
External Interface Testing
Overall Project Plan & Status*
*includes current architecture, design, documentation, released code, test strategy, & project plan,

Figure 5. One Cycle in the Evolution Phase: Additional Detail

Lessons learned in any cycle may lead the team to revise the overall plan developed in the previous (Conceptualization) phase. In this case, it is important to focus on the project goals and carefully evaluate the benefits of the changes versus the costs of not making the changes. Find the correct level of change for success while keeping the project on track and avoiding “requirements creep”.

5.4.1 Objectives of Each Evolutionary Cycle

The principal objective of each cycle of the evolution phase is to complete the interim products and deliverables that meet the planned goals for that cycle.

Deliverables for each cycle include software, test cases, and documentation. All should be placed under configuration management. To complete a cycle, the software delivered should have completed unit testing, integration testing and validation and evaluation by users or customers as appropriate. Documentation produced in previous phases or evolutionary cycles should be updated to show all revisions and additions. This will include at a minimum the functional requirements, system architecture and overall project plan. As the system evolves, design and implementation details should also be captured in as-built system design documentation. Other documents, products and deliverables may be required, as called for in the overall implementation plan and the detailed plan for the cycle.

Each cycle should conclude with a report which details the results of that cycle. The report should specifically address the planned objectives of that cycle. Were the objectives of the cycle met? How, or why not? What alternatives are available for missed objectives? The report should include response to user input from the previous cycle. This could include design or implementation changes, cost/benefit trade-offs, actions taken, results, or other responses. A user evaluation for this cycle should also be included.

Depending on the plan for the cycle and the results achieved, other information may be appropriate to include in the cycle report. It could address design constraints that were included, along with rationale. Where alternatives were previously identified, the report should indicate which alternative was selected, the selection criteria used, and the implications of the decision on this and other systems.

Other issues which need to be addressed, as appropriate, include:

- risk resolution/results
- schedule impacts, modifications
- cost
- deliveries
- test results

Before completing the cycle, the project team should reassess feasibility in light of results achieved in this cycle. A detailed plan should be prepared for the following cycle. Lessons learned in this cycle, as well as plans for the next cycle, may impact the overall implementation plan. If this is the case, then the implementation plan must be updated as well. Update other documentation, such as system architecture and as-built design documents, as appropriate.

The first cycle should emphasize system architecture over detailed functionality. It forms the backbone for the evolving system, and should include end-to-end integration of all planned system modules and their interfaces, both internal and external to the project. At the completion of cycle one, many of the modules may have little functionality except as place holders, and interfaces may be vague, but all major system components and the way they fit together should be well understood.

Cycle two should emphasize system utilities, project development standards, and analysis of available code for reuse.

Appropriate levels of formal testing, verification and validation should be included in each cycle. A careful balance must be struck between complete testing, superfluous testing, and redundant testing of each cycle release. For example, it is probably a waste of time to completely test detailed functionality of a low fidelity early release. There may not even be detailed specifications for low fidelity interim models, since they are planned to evolve to the final system. Also, some modules may change little from one cycle to the next. For this case, full scale testing may be redundant. Still, each release forms the basis for the next, so formal testing is required to assure that the basis is complete and accurate. Significant engineering judgement is required in deciding test requirements. For real-time systems, timing issues should be addressed in each cycle’s test plan.

In this environment, developers will typically have powerful simulation and test capability available in the development environment. When this is the case, unit test drivers, data, and results can often be delivered with the code. In this case, it is appropriate for formal testing to primarily emphasize module, subsystem, and system interfaces, rather than unit functionality.

Formal code inspections of the evolving system are recommended. Since this methodology establishes low-fidelity end-to-end functionality early in development, and then evolves the modules to full detail over several releases, it is generally not reasonable to inspect all modules for all releases. A module should be inspected when it has reached a maturity level that includes substantially all of its expected functionality, or earlier if the developer of that module requests it. And modules should be reinspected if substantially modified for any subsequent release. It is appropriate to rely on the best engineering judgement of the developers to make the qualitative decisions on what constitutes “substantial”. For modules developed with graphics development tools and auto-code generators, inspections should be performed on the block diagrams rather than the generated source code.

Augment the core team to include users/operators of the system as appropriate, but especially in the design, prototyping and inspection of user interfaces and displays.

When possible, as well as available, the use of advanced software development, test and quality assurance tools can further leverage productivity and quality, especially in this phase. If such tools are used, it is important to assure that sufficient licenses and platforms are available to the team, so as to avoid resource bottlenecks which may impede project momentum.

Rarely do projects proceed independent of outside schedule pressures. These may be motivated, for example, by inter-project dependencies or by the need to demonstrate interim releases. Care should be taken to include these dependencies in the evolutionary plans, to avoid significant schedule impacts. Plan for the need to work higher levels of integration and test, and for the (likely) resulting rework. This is especially important when the final system will integrate multiple applications developed by multiple organizations.

Migrate new work to reuse library, as appropriate.

There are some questions still to be resolved about the best approach for independent V&V of early releases. Full IV&V of each release is likely to be expensive, time consuming and overkill, and too much close cooperation with the development team could compromise some of the objectivity of the IV&V team. But there are also some advantages to involving the IV&V team early. Consistent with the Rapid Development philosophy, any problems which the IV&V team can identify early should be easier and cheaper to fix, compared to finding the problems after development is completed. By working with the IV&V team early, developers may become more knowledgeable about the testing capability desired by the IV&V team and be able to build it into the system; this could later facilitate an easier IV&V effort. And by working with early releases, IV&V team members may be more familiar with the system and able to proceed faster with final IV&V.

5.5 High Level Objectives of Finalization Phase

This phase is really just the last planned evolution cycle, but there should not be any remaining issues when this cycle completes. Note that when using evolutionary and prototyping techniques to speed system development, performance tuning and stress testing often are quite critical activities at this point in the development.

Primary elements of this phase include:

- Final Performance tuning
- Stress testing
- Finalize documentation
- IV&V
- User, customer and developer sign-offs
- Establish plans for user support, maintenance and upgrades
- Installation and Transition plans
- Resolve remaining discrepancy reports
- Archive final test results
- Complete migration of identified modules to reuse library
- Document all known successful configurations (e.g., processors, operating systems, auto-coded languages, simulation and test venues, etc.) and instructions for building the system in the various configurations

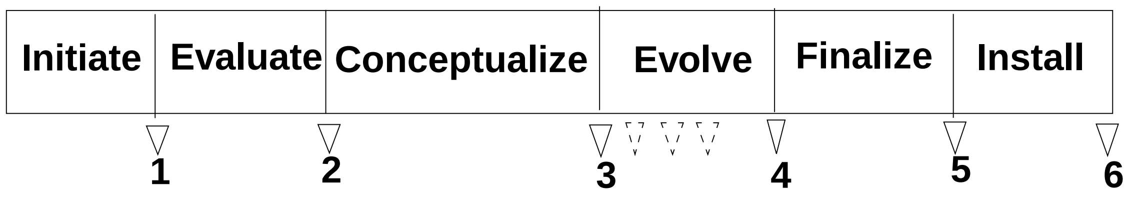
5.6 High Level Objectives of Installation Phase

In the Installation phase, the system is made available for its intended use. Activities include setting up scripts and procedures for everyday use of the system. User support, maintenance and upgrade plans, from the previous phase, should be initiated. Support for user training and start-up activities is required.

Since at least some of the user community have been involved in the development process, the Rapid Development methodology should facilitate smooth installation of the completed system.

5.7 Summary

The figure below restates, in summary fashion, the major phases of development under the proposed Rapid Development life-cycle model. The previous sections have discussed primary activities of each phase. Some of the most important of these are called out again in the figure. To support these activities, key software engineering support processes will be outlined in a later section.



1: Statement of problem to be solved; Decision (continue project or not)*;
if project to be continued: Composition of design team; Commit Budget for next phase.

2: Feasibility & Risk Assessment report; Development Strategy selection

3: Overall project plan, including reuse, software engineering tools, hardware, build strategy; Detailed plan for cycle 1; High level system architecture prototype;

4: After each cycle: Status wrt current cycle plan and overall project plan; Detailed plan for next cycle; Updated project plan

5: Final Documentation; Sign-offs; Installation & Transition plans; Maintenance & support plans

6: Installation completed; Start-up completed; Training completed

* This decision point is implied for each phase, even though not explicitly stated.

Figure 6. Project Management Milestones

6.0 The Maintenance Phases of the Life-Cycle Model

In the most part, the maintenance phases of this lifecycle model will be quite similar to traditional methods. They will differ primarily in the emphasis on life expectancy evaluation (for long term planning), in the Sustaining Engineering phase, and reuse consideration in the Shutdown phase.

6.1 High Level Objectives of Sustaining Engineering

The principal objective of Sustaining Engineering is to maintain system usefulness by protecting system integrity, fixing problems that are identified, and performing modifications to keep up with changing environments. Documentation management, including updates and configuration control, are part of this effort.

Several secondary activities support the primary objective. These include supporting user activities, providing user help support, and user training.

On a regular basis, the system should be evaluated to ascertain remaining life expectancy of the system. The evaluation should include some analysis of the cost of maintaining the existing system versus the cost of replacement. If the need to replace or significantly upgrade the system is anticipated, then it is desirable to include an estimate of lead time required and potential cost. Keep a running list of prioritized documented potential upgrades to use as input to any upgrade projects. Plan and lobby for replacement, if needed.

6.2 High Level Objectives of Shutdown

If the system is determined to have reached the end of its useful life-cycle, and if any necessary replacements have been installed, then an orderly shutdown is called for. This could include:

- Verify that all relevant elements have been migrated to the reuse library
- Archive software, documentation and hardware
- Release licenses
- Surplus hardware
- Assist users with migration to new system and procedures

7.0 Key Support Processes for Rapid Development

In many ways, the key support processes for Rapid Development are similar to those used in traditional development paradigms. This section will therefore briefly introduce the concepts and highlight some suggested modifications for Rapid Development.

7.1 The Need for and Application of Support Processes

Support processes are intended to control the development process in ways that improve chances for success. That is, the support processes are put in place to ensure that the system is developed correctly, on time, of adequate quality, and on budget, that the documentation is complete and the code is safely stored and retrievable, and that management is informed and aware of progress, problems and results of a project.

Support processes complement the life cycle, provide feedback to management on the progress of development, and provide information which can be used to drive process improvement.

As with the life cycle, the particular support processes used and their implementation should be customized specifically to best support each particular project. Moreover, the customization process should take into account the skills and experiences of the project team, taking advantage of any history and expertise with specific products, techniques or processes.

In choosing and implementing support processes for a project, the following issues and questions should be considered:

- How will the technical process be controlled?
- How will the use of resources (staff, budget, equipment) be budgeted, tracked, and controlled?
- How will project planning be done, both initially and in response to actual progress and status?
- What are the key risk areas for the project? How will they be identified, tracked and controlled?
- What are the key data products of the project? How will the data products and software products be managed and controlled?
- How will document content be managed and controlled?
- What tools and equipment will be used for this project? Include tools and equipment used both for development and for support processes.
- How can the system development process be measured to identify quality, cost, and schedule status and issues?

The answers to these questions will help determine what types of support processes are required for the project and how they should be implemented.

7.2 Types of Support Processes

One way to characterize system development processes is as either technical, management or institutional processes. In this view, the technical processes support completion of the tasks needed to perform a project, management processes support the tasks needed to monitor and control progress and resources for a project, and institutional processes support the tasks needed to maintain the organization and environment in which a project takes place. The following sections will explore each of these process types in more detail and discuss key issues and questions to consider when setting up support process tools and procedures for a project.

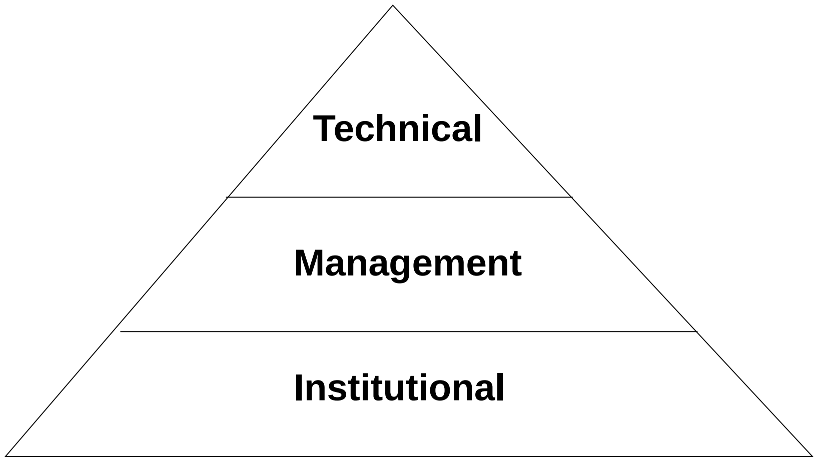


Figure 7. The Support Processes Pyramid

7.2.1 Technical Processes

Technical processes produce the product. These include all the steps in a project life cycle (discussed in some detail in previous sections of this guidebook), along with test and V&V (verification and validation) processes.

Management of the technical process includes monitoring technical issues with respect to the project plan. Experience shows that the exact method for doing this is highly individual, depending on the style of a particular project leader and that of the team members.

There are some key ideas to keep in mind when doing technical management in a Rapid Development environment. To achieve maximum success, flexibility and responsiveness must dominate project tracking. A strong project leader should as much as possible anticipate problems and have alternatives identified. Team members should quickly inform the project leader of any difficulties with potential schedule impacts. Frequent replanning will usually be required. Technical management of Rapid Development projects is a high energy, highly interactive process. To be most effective, the project leader and team members should have considerable authority to revise, rework, and reassign tasks, priorities and resources as needed to meet deadlines, budgets and requirements.

Team communication is a key element for success. All team members need access to the latest plans, schedules, requirements, priorities and decisions. There are many ways to achieve this, from a centrally located notebook that is updated frequently, to on-line web pages accessible by the whole team. Frequent status tag-ups can be useful, but are not a substitute for written material.

Frequent technical status meetings can be quite useful if they are short and focused. Strong technical management should guide these meetings to assure that needed information is exchanged but details not of interest to the entire team are worked independently. Avoid the trap of over discussing issues by having a clear understanding of who has decision making authority; it is not necessary to achieve consensus in every area. It will generally improve the project to solicit team input for central issues or especially difficult design areas, but too much discussion can impede progress.

Problems which may result in significant modification of the plan, whether in delivery content, milestone dates, or resource usage, should be communicated to management in a timely manner. Keep in mind that management abhors surprises, and can support the team better if status and needs are clear and up to date.

7.2.1.1 Inspections

Code inspections are highly recommended. Inspections have proven to be a successful mechanism for identifying and eliminating errors early.

Some of the important aspects of a successful inspection process include:

- *Focus inspections on project improvement.* Inspections are not personnel evaluations. Managers should not be involved, and inspection results should never be used in any way to measure individual performance.
- *Each inspection meeting should be short and focused.* Meet for no more than two hours. Try to select small (in the range of 200 SLOC or 40 blocks) sections of code for each inspection.
- *Inspection teams should be small and focused.* For each inspection, the team should include a facilitator, presenter, recorder, developer and inspectors. Often, these functions will overlap (for example, the developer may also present, an inspector may also record). A typical effective inspection team size is four to six people.
- *Inspections must focus on identifying problems and their severity.* If team members need to be briefed on the code to be inspected, hold a pre-inspection briefing meeting. If the developer is uncertain how to fix a problem, address it outside the inspection meeting. During the inspection, concentrate on creating a complete list of defects.
- *Prepare for the inspections in advance.* The facilitator must verify that the developer has the code ready. Together, they must provide the rest of the inspection team with code, documentation and any required supporting material, several days before the inspection. Team members should review materials and prepare a defect list prior to the inspection.
- *Record inspection results.* This includes listing all defects identified, the severity of each defect, and any follow up actions required.
- *Verify that the defect list and action list have been successfully addressed.* Usually the inspection facilitator is responsible for this, and works it directly with the developer, including others as needed. Follow up meetings are not usually required.

Some special aspects of successful inspections in a Rapid Development environment include:

- When graphical development environments and autocoders are used, it is appropriate to inspect the block diagrams rather than the generated source code. This has proven effective for several reasons. Auto-generated code is usually difficult to read. If defects are identified, any changes to the module should be made to the block diagrams anyway. (This is in order to preserve the ability to maintain the code via the graphical language. Anytime the auto-code is changed manually, it is no longer represented by the block diagrams.) Since the block diagrams are often easier to read and understand than source code, it is easier to include domain experts (who may not be coding experts) in the inspections.

- Do not inspect every module for every release. The Rapid Development method establishes low-fidelity end-to-end functionality early in development, and then evolves modules over several releases. It is usually not effective to inspect very low fidelity models or incomplete modules. A module should be inspected when it has reached a maturity level that includes substantially all of its expected functionality, or earlier if the developer of the module requests it. Also, it may be appropriate to reinspect modules that undergo substantial revision for a later release.
- Do not delay inspections. It is tempting to wait until the system is completed before performing inspections, but this is not optimal. By inspecting some modules early in the development, coding standards, project expectations, and developer understanding will be enhanced. Also, inspections are demanding and time consuming. Team members can give them better attention, and produce better inspection results, if they are spread out across development rather than concentrated at the end.

7.2.1.2 Leveraging Project Information

Any project produces lots of paperwork. We try to capture information about requirements, design, plans, schedules, progress and status. Use technology to enhance the value of this information in a Rapid Development environment.

Some useful approaches include:

- Use electronic documentation. Try to find and use a relatively standard format (today, this might mean PDF or HTML, but this could change rapidly). It is almost always faster and cheaper to produce, maintain and deliver documents in electronic form.
- Make the documentation accessible to the project team. Use the network. FTP or web sights are appropriate now; other technologies may emerge. This assures that all team members have access to the latest versions of all documents. It means that the documents still are available when different work stations are used. It eliminates sources of potential error, confusion or inefficiency.
- Link the information. Requirements inspire design. Design relates to implementation. All correlate to test plans and test results. With existing technology it is possible, and not usually very difficult or costly, to electronically link the information, and even to automatically process it to get status snapshots. A Rapid Development project will be well served by careful and automated information correlation.

See section 7.2.2.6 for additional insights into documentation issues in a Rapid Development environment.

7.2.2 Management Processes

Management support processes are put in place to monitor and measure progress, while providing resources and support to the technical project team.

Key management processes generally include Resource Management, Project Management, Risk Management, Configuration Management, Test Management, Data & Documentation Management, Problem Reporting and Resolution, and Information Resource Management.

7.2.2.1 Resource Management

Resource Management includes tools and procedures used to budget, track and control the resources that are available to projects in an organization. Typically these resources include staff, budget and equipment. The primary issues to be addressed by resource management are:

- What resources does the project require? For each required resource,
  - When in the project life cycle is the resource required?
  - Must the resource be dedicated to the project or can it be shared?
  - How does the commitment required of the resource vary as the project progresses?
- What resources are available to the project to fill these requirements?
  - Are all the required capabilities represented in sufficient quantity?
- How does resource availability correlate to resource requirements during the project life cycle
  - There may be a need to coordinate with other projects.
  - This will affect project schedule.

Like most aspects of any project, and especially those using Rapid Development techniques, the resource requirements profile is likely to vary from the plan somewhat as the project progresses. It is important to track the variations and project their implications to ensure that resource allocation plans are still feasible. It is usually counterproductive to insist on exact balances in the resource allocation, since requirements are based on estimates which will vary somewhat from actuals anyway. It is more useful and important to plan for approximate balance, and then monitor and revise plans frequently as necessary.

Effective resource management in a Rapid Development environment can be challenging. A Rapid Development approach to systems implementation may require more detailed resource management than traditional models. This is because skill requirements may overlap among more phases of the system life cycle than the more partitioned classical approaches.

7.2.2.2 Project Management

Project Management includes the tools and procedures used to plan, schedule, and monitor project progress.

Project planning involves identifying the tasks that need to be done, and any interdependencies. Estimate the resources (time, staff, equipment, budget) required to perform the task. Determine significant milestones, deliverables, and any entry or exit conditions for the tasks.

The project schedule assigns resources to each task and lays the tasks and milestones out on a timeline.

Project management procedures and project progress will determine how often actual progress will be compared to planned progress and how and when the plan and schedule will be updated.

There are many things that cause actual progress to deviate from plans and schedules. A task may prove to be harder or easier than expected. Resources may not be available as expected, due to procurement difficulties or unexpected requirements from other projects. Internal or external ICD (Interface Control Document) conditions may not be on schedule. New tasks may be identified that are required but were not previously in the plan. The general expectation is that the near term portion of schedules will be more accurate than the long term portions of schedules.

In the Rapid Development environment using an evolutionary/spiral implementation model, an overall project plan and preliminary schedule should be prepared as part of the Conceptualization Phase. This high level plan will include an outline of the planned evolutionary cycles and goals for each, including entry and exit conditions. At the same time, detailed schedules for the first evolutionary cycle must be prepared. Since the detailed plans for each succeeding cycle will depend somewhat on the lessons learned in the current cycle, detailed scheduling will often be deferred for a cycle until near the end of the previous cycle. Yet a best estimate look at plans and schedules for the entire project is highly beneficial, in order to obtain and commit the necessary resources, to know when the system will be available, and to know when the team and other resources can be used for other projects.

7.2.2.3 Risk Management

Risk Management processes are used to assess, control and minimize project risk. Risk areas for a particular project could include cost, schedule, quality, safety, security and feasibility.

The nature of the Rapid Development life cycle introduces some special risk areas for a development effort. This is primarily due to the evolutionary nature of system requirements and project plans in this environment. Some of the special issues to be aware of include:

- Requirements Creep. As the life cycle progresses, more becomes known about the problem and its solution. When using Rapid Development, requirements discovery is a natural part of each development phase. As users and customers begin to see and use the results, the project team will naturally think of additional capability and new ways to use the system that are desirable but may be outside the original scope of the project. Since each cycle and phase can legitimately result in modifications to the requirements, there is a risk of constantly increasing the scope of the project. This, of course, can result in missed deliveries and cost overruns. Yet refusing to respond to changing requirements can negate some of the benefit derived from the Rapid Development environment. To solve this dilemma, the team must stay focused on the scope of the project. New or modified requirements requests should include an indicator of their criticality and estimated costs. When changes to the requirements are likely to result in changes in the agreed costs or schedule, management and customer agreement should be obtained. In formal environments, such as support contractors, this agreement should be in writing and should include the relevant contracts personnel. Sometimes desirable capability will not be implemented or will be deferred to a later project. Try not to let this reduce the dynamics and enthusiasm of the team; identifying and documenting ideas for future implementation is valuable, and delivering a good system on time and in budget for the current effort increases the chances that additional work will be funded later.

- Inconsistent expectations. The nature of the Rapid Development environment is that things change quickly and agreements tend to come in meetings or brainstorming sessions. But we all know that different people can come away from the same meeting with quite different ideas about what was agreed upon. Take the time to summarize and distribute updates with key points, and verify agreement among interested parties, no matter how great the temptation to “just do it” may be. Rapid Development techniques may skip some of the traditional upfront documentation and review steps, but still requires that major requirements and design decisions be recorded and approved as they evolve with the system.

Some aspects of the Rapid Development model tend to reduce project risk. Typically, plans for the evolutionary cycles will identify the highest risk areas of the project and work those first. After the solutions to the hardest problems have been determined, designed, and implemented or prototyped, the lower risk, easier, better understood problem areas can be addressed. It should be relatively straightforward to adapt the low risk system functions to the (then) in place solutions to the high risk problems.

The Rapid Development life cycle also encourages risk containment by:

- limiting the work that must be done before system development begins
- involving users and customers in development decisions, to improve user acceptance of systems
- frequent replanning improves flexibility to respond to changing requirements and budgets
- interim cycle deliveries contrast with the “all or nothing” mentality of traditional development and, in the face of budget uncertainty, ensure that completed work is captured in usable form at predictable points in the project

7.2.2.4 Configuration Management

Configuration management systems and procedures (CM) define, implement and enforce the ability to track project information.

The most common application is software tracking. The configuration management process maintains the official software and tracks changes to it.

A complete configuration management system will usually include additional information. Common elements include documentation, test cases and results, and system support specifications (such as compiler requirements, library dependencies, and hardware and operating system configuration needed to support the release).

There are some special considerations for configuration management in a Rapid Development environment.

Usually there will be several official releases of the system (at cycle intervals). At least for the duration of the development phases, configuration management should track and support each release. This means freezing a snapshot at release time, and maintaining access to that snapshot, and previous snapshots, while development of succeeding releases progress.

Each official release should be linked (preferably electronically) to documentation that also corresponds to that release, such as requirements and design documents. Capturing the detailed plan for the cycle is also recommended, since this will give information about the goals and capabilities of that release.

Test data, test drivers and test results are important elements to capture, place under configuration management, and link to managed code. Also, system configuration requirements (hardware, operating system, compiler used, etc.) should be recorded for each release.

Whether there is a need to maintain interim versions at the conclusion of the development effort is project dependent, and the CM should include decision checkpoints for this.

The configuration management process must include mechanisms and controls for evolving between cycles. A variety of COTS tools are available to support check-out, modification, and check-in of files.

Historically, configuration management (CM) has been used as a gateway to make sure that only approved software passes into the system. Typically, changes made to software under CM had to be correlated to change requests (CRs) or discrepancy reports (DRs) and receive management or board level approval before being accepted.

This sort of system may be too rigorous for early Rapid Development evolutionary releases. Thus, teams may be inclined to delay putting the software under configuration management until it is nearly completed and has begun to stabilize, so as to avoid interfering with the momentum of Rapid Development in the early cycles.

But that approach does not adequately protect the project. In Rapid Development, changes may occur frequently and various developers need to coordinate to make sure that everybody is using and testing with the current version. It is also necessary to verify that changes made by more than one developer do no conflict or cancel each other.

The recommended configuration management approach for Rapid Development is one that is more open to the developers, somewhat less rigorous in terms of control, but quite powerful in terms of tracking changes, integrating work of multiple developers, and coordinating multiple releases. A variety of COTS tools exist that can support such an environment.

For early development cycles, it should be relatively easy for implementors to check out, modify, and check in software, using the CM system. This places relatively little demand on developer productivity, while providing excellent protection of the evolving software. As the software matures, and passes through more testing and approval gateways, control over software modifications should be increased.

Configuration management in the early stages of Rapid Development can be used effectively to the benefit of projects. It is important to strike a balance between control and protection of project assets, while maintaining enough flexibility to support a highly dynamic work environment. This type of intermediate and flexible control can be achieved with currently available support software.

7.2.2.5 Test Management

The Test Management process defines required testing procedures, the process for ensuring and documenting that all required tests have been performed successfully, and the conditions that require repeating some or all of any test series.

When working in a Rapid Development environment, several deliveries (cycle drops) are usually planned for a project. Complete, independent verification and validation (IV&V) is often done only in the finalization phase. Unit testing, subsystem testing and integration testing is needed for each evolutionary cycle. At the completion of each development cycle, sufficient testing should have been completed so that the team is confident that the release works correctly or has documented known problems and included problem disposition in the project and cycle plans.

Specific test requirements should be included in the detail plan for each cycle. It may be necessary to repeat some tests from previous cycles to be sure that the current cycle's updates have not inadvertently caused problems in previously completed work. Other test runs from previous cycles may no longer be appropriate or may require modification to reflect the current state of the system.

As much as possible, the test process should be automated, to save time and to simplify repetitive and repeatable testing. Coordinate with Configuration Management to archive test cases, drivers, results, and interpretation. Link with electronic documentation to maintain the relationship among requirements, tests and results.

After development has completed and the system has been delivered, it may not be necessary to maintain interim cycle test information. This issue is highly project dependent. Consider whether any of the intermediate cycle deliveries might serve as starting points for related or future projects. If so, the entire test suite may be valuable. Will any modules migrate to the software reuse library? In this case, certainly unit test data and drivers will be useful.

7.2.2.6 Data & Document Management

Data and Document Management processes define data and document requirements for a project, as well as responsibility for their creation, procedures for approval and distribution, procedures for archiving, maintaining and updating project data and documentation.

A Rapid Development environment creates some non-traditional issues in this area, especially documentation. Specific documents to be written, and their timing with respect to the project life cycle are often markedly different from more traditional models. Subsystem design documentation usually follows or parallels implementation. Requirements documentation tends to be more functional than detailed and may evolve with the system. The implementation generally serves as the detailed design, especially if graphic development tools (with auto-coders) are used. Updated documents may be released with each implementation cycle.

Released documentation should be held under configuration management. Until project development is completed (and, presumably, documents are in their final form), it is probably best to limit distribution of the documents. Accessible electronic copies of documents are recommended, since they allow access as needed to the most current versions. Excessive distribution of interim documents runs the risk of confusing or overwhelming recipients who receive copies of several versions. Interim versions should be clearly marked as such, to minimize confusion.

If some form of electronic approval capability (equivalent to signatures on a hard copy) could be implemented, it would be possible to also maintain the “official” copies electronically.

Whether or not to maintain availability of interim versions should be decided on a per project basis.

7.2.2.7 Problem Reporting and Resolution

Problem Reporting and Resolution processes define and carry out the formal mechanism for logging observed problems, facilitating their resolution and tracking their status.

Problem Reporting and Resolution processes are used during testing and during the sustaining engineering phases of a system life cycle. Traditionally, reported problems have been categorized as

- discrepancy reports(dr): documents an aspect of the system which does not match stated requirements
- change requests (cr): documents a need to modify one or more stated requirements and the system
- trouble reports(tr): documents a general problem in operating the system

In the Rapid Development environment, during system development phases, requirements are expected to evolve with the system, so that one end product of each evolutionary phase is updates to the system requirements. These may be additional requirements, or clarification of general requirements previously outlined in the requirements documentation. As these requirements are identified, they need to be logged and tracked as well. These may be included in the problem reporting system as a fourth category, requirements change.

- requirements change (rc): documents a newly discovered requirement or a clarification of a general requirement

Note the difference between a cr, which documents a change to existing requirements, and rc, which documents a newly discovered requirement.

Disposition of requirements change reports must be carefully managed to avoid the problems of requirements creep (see section 7.2.2.3, Risk Management). Problem reports should be linked to requirements and test results (both the results which identified the problem and the results which support problem resolution). Problem summary reports, which summarize status and disposition, should be automated as much as possible, and available to team members.

7.2.2.8 IRM (Information Resource Management)

Information Resource Management deals with issues pertaining to tools, systems and equipment that will be used for a project. These could, for example, be data base tools, compilers, CASE tools, development environment, computers, networks, operating systems, and office support tools such as word processors and spreadsheets programs. An information resource management plan works to ensure that the necessary resources are available to support the project. For small projects, the plan is often unwritten and relatively informal. More complex projects may require more formality in managing these resources. In any case, it is important to investigate and plan for the availability and stability of the information resources that are required by the project.

Project plans and schedules can be effected by the information resource management plan in a variety of ways. If there are a limited number of licenses or workstations, there may be competition among team members or between teams for those resources. If a piece of software or hardware undergoes an upgrade there may be conversion requirements or down time interruptions of the schedule. There may be a need for technical support, both internal (from your organization) and external (usually the product’s vendor), and the project budget must account for this. New or upgraded tools may necessitate training time and costs, or may have a “learning curve” effect, temporarily reducing productivity. System Administrators may be willing to accelerate or delay installations or upgrades to facilitate a project, or there may be conflicting requirements among projects that need to be negotiated.

Of special interest in the Rapid Development is the effect of the evolution of the information resource environment and its effect on intermediate deliveries. For example, if a compiler is upgraded, will there be a need to test interim releases of the system, or only the version currently evolving? Or, what happens to electronic project records (documentation, meeting notes, etc.) if the document preparation package changes? The answers to such questions are, of course, project dependent. It is important to be certain that they are considered.

Powerful tools are likely to be in high demand by the implementation team. Plan ahead to assure sufficient availability of product licenses to support the project needs.

7.2.3 Institutional Processes

Institutional processes are not project dependent, but rather project supportive. That is, these are processes which are put in place to generally improve the ability of an organization to develop good, cost effective systems in a timely manner. Some important examples of institutional processes include Labor Accounting, Process Improvement, Staff Training, Tool Evaluation & Selection, and Metrics Data Collection, Evaluation and Reporting.

7.2.3.1 Labor Accounting

A labor accounting process provides the ability to measure and track labor costs. Labor accounting may be done at varying granularity (minutes, hours, days), and detail levels (project, phase, task, department) depending on the needs of management.

Some examples of the uses for labor accounting data include:

- to monitor progress by showing what is being worked on
- to compare estimated to actual effort required to complete a task
- as historical data to improve future estimation capability
- to determine project costs, for internal accounting or customer billing
- to determine productivity trends and other project management metrics

7.2.3.2 Process Improvement

How does an organization know if its processes and procedures are effective and efficient? Process Improvement processes examine current ways of doing business, identify areas of weakness or potential improvement, then propose, implement and evaluate new methods as appropriate.

ISO-9000 and the SEI (Software Engineering Institute) CMM (Capability Maturity Model) are two approaches to process improvement that are widely accepted.

The Rapid Development guidelines in this document are a major effort at overall process improvement. A correlation between the Rapid Development methodology and SEI's CMM is available in JSC-38607.

7.2.3.3 Training

In addition to project specific training, there is a general organizational need to keep staff up to date on current tools, processes, technology, etc.

When an organization starts using the Rapid Development model, project staff may not be familiar with Rapid Development the techniques and tools. Start up training will be necessary.

7.2.3.4 Tool & Equipment Evaluation & Selection

What is the best way to equip a work area? Tool and Equipment Evaluation and Selection is an ongoing, continuous process, because the state of the art advances, new products are offered, current products are upgraded, equipment wears out. Planning and budgeting for this activity will help ensure its success. If possible, survey staff for product or capability wish lists, and then keep a lookout for them in the marketplace. Document product evaluations for use by others and comparison with other products. If possible, share evaluations with other organizations, to get alternative views and to increase the possible scope of the evaluations.

Tools which support Rapid Development proliferate. To keep on the cutting edge, active attention to their evaluation is recommended.

As selection and procurement are planned and carried out, coordinate with training efforts. Consider effects on project plans, schedules and budgets (see section 7.2.2.8, Information Resource Management)

7.2.3.5 Metrics Data Collection, Evaluation and Reporting

Metrics related processes pertain to the collection, evaluation and reporting of project data.

Metrics can be used to support project development, maintenance and management functions, as well as process improvement functions. It is an Institutional process because of the advantage to an organization of using consistent tools and procedures for metrics across projects. A consistent approach allows for project to project comparisons and improves the data collection results, as staff become familiar with what is expected and adopt it as part of the normal work environment.

Expect an effective metrics program to take two to four years to mature. Look for balance in the amount of data collected versus the effort needed to collect it. Automate the collection and reporting process as much as possible.

Evaluation of metrics can sometimes lead to unexpected results. Be open to new possibilities. Expect some initial staff resistance to the concept; acceptance will likely follow once the value of the program is demonstrated.

When choosing what data to collect and how to evaluate it, focus on the goal of metrics. The easiest data to collect is not always the most useful. Productivity can be difficult to quantify, especially in high tech environments where one-of-a-kind systems are developed with advanced tools, such as auto-coders. Try not to overlook, for example, the advantage that would be obtained by improving estimating techniques or reducing error rates in delivered systems.

Effective metrics processes for a Rapid Development environment are still being researched. The next section of this document looks at additional issues regarding the use of metrics in a Rapid Development environment.

8.0 Using Metrics for Success in Rapid Development

Metrics are collected in order to measure, track, manage and plan projects. The successful use of metrics depends on knowing what to measure, taking accurate measurements, and interpreting those measurements. Much of this is predicated on having historical data for comparison. For Rapid Development, there is very little available historical data, including the large commercially available project metrics data bases. We, along with others in the software systems field, are developing appropriate methods as we go, while trying to take what we can from metrics programs that have been developed for traditional methodologies.

8.1 Metrics Goals

Metrics can be useful only insofar as they support project goals. In order to determine what to measure, it is necessary to consider the goals of the metrics program. Some of the things that a manager or team member might want to know about a project, and might use metrics to help discover, include:

- predictive estimates of, for example
    - staffing
    - time
    - cost
    - system size
    - system performance
    - benefits
    - risks
- project output
- project outcome
- status
- progress
- quality

A particular project may need a metrics programs to determine some, all or none of these, or other project evaluations.

8.2 Measuring Values for Metrics

Every project will have individual needs. For traditional software development efforts, there is significant historical data available to provide baselines of what to measure and how to use those measurements. These ideas are less well researched for Rapid Development.

On some level, it may be appropriate to measure whatever is available. In the Rapid Development Laboratory, we are attempting to build our own data base of historical measurements so that, over the long run, we can gain better understanding of the appropriate use of metrics for Rapid Development projects.

But each measurement taken has some impact on the projects, at least in time to collect, store and study the measures. So some initial analysis is required to focus the measurement effort.

Several strategies have been investigated.

One effective metrics program in use at NASA, in a traditional development environment, was adopted by the JSC/Mission Operations Directorate (MOD) during the period from May 1990 to March 1992. This metrics program has been and is being used successfully as a management tool on The Mission Control Center upgrade task at JSC and the development of the Integrated Planning System for Space Station planning and analysis. Both these projects are large efforts requiring incremental development to manage risk and respond to fluctuating budget uncertainties. Details pertaining to this program are included for reference in Appendix D. This program was studied extensively by RDL staff before beginning our own metrics program.

In a cooperative technology exchange program, our initial metrics efforts were shared with and studied by the Goddard Spaceflight Center (GSFC) Software Engineering Laboratory (SEL), in cooperation with the University of Maryland. In this effort, our initial metrics program was analyzed in the context of a GQM (Goal/Question/Metrics) program being studied at the SEL. This effort is still underway.

Table 1 shows the metrics currently being collected in the RDL. Table 2 shows the preliminary SEL suggestions for data collection using the GQM approach in our environment. Neither list should be taken as definitive. This is an area that requires additional research, and these are preliminary efforts. In particular, the SEL/Univ of MD study is still underway. They are presented here in the hope that they may prove to be useful starting points for a RD metrics program.

Table 1. Initial Metrics Collected

Metric Classification	Description
Software Size	The SLOC in the system that must be tested and maintained
Software Size	MatrixX Block Counts
Software Size	Size of executables
Software Staffing	Number of engineering and first line management personnel involved in system development
Development Progress	Number of modules successfully completed from design through test
Software Performance	Execution times
Test Case Completion	Percent of test cases successfully completed
Discrepancy Report Open Duration	Time lag from problem report initiation to problem report closure
Fault Density	Open and total defect density over time
Design Complexity	Number of modules with a complexity greater than an established threshold

Table 2. GQM Metrics to Support RD (preliminary)

Actual expended effort by time period and subsystem
Expected effort by time period and subsystem
Total effort
Effort by cycle
Effort by WBS (Work Breakdown Structure) task
Number of defects discovered, by date
Number of defects expected, by date
Number of defects fixed, by date
Number of defects discovered, by subsystem
Number of defects discovered, by test activity
DR (discrepancy report) open duration
Number of severe defects discovered
Total calendar time
Calendar time by cycle and task
Number of cycle and WBS tasks completed to date
Number of cycles and WBS tasks scheduled to be completed
Size of system in blocks, units, principle functions and/or requirements

For some of the measurements in both tables, we do not yet have sufficient historical data base or insight to know how or if they will prove useful for planning or managing a Rapid Development project, or for evaluating its quality. An example is MatrixX block counts. By collecting this data, we hope to learn more about effective metrics in a Rapid Development environment.

8.3 Metrics Insights for Rapid Development

This is a preliminary effort. There is not yet sufficient data or experience to draw significant conclusions about the best way to use metrics in Rapid Development. We are still looking primarily at output metrics, which monitor and record project progress, as is appropriate for an early effort. As we learn more, we hope to be able to concentrate more on outcome metrics, which can be used to position a project for success more pro-actively.

Our experiences to date have yielded some important insights. These include:

- Specific development resource utilization metrics are not only difficult to collect but probably irrelevant in our development mode. CPU time, for example, is not a typically scarce or limiting resource. And those resources that are limiting are usually obvious: insufficient licenses for some development tools or test platform availability are common resource bottlenecks. When we get more sophisticated at this, it may be useful to revisit this type of metric, to help determine, for example, how many additional licenses are optimal.

- A more relevant and critical resource measurement appears to be the performance of the developed software. In the RDL, this is due to the real-time demands of GN&C systems.
- Code size measurements, whether as SLOC or MatrixX blocks, are not predictive of project progress for Rapid Development, since the code tends to expand rapidly in early iterations and then stabilize during cycles which refine functionality. Code size may still be a good predictor of cost or schedules, so we choose to continue to measure it to populate our historical data base.
- Function Point measurements should be further investigated, but require significant training and understanding for successful use.
- Automated, semi-automated, or on-line tools can be developed to both improve the accuracy and currency of data and minimize the impact, on team members, of capturing the data. When such tools are used, they need to be captured under configuration management, to assure consistent measurements throughout the project life-cycle.

8.4 Some Special Aspects of Rapid Development Metrics

When designing and implementing a metrics program for Rapid Development, consider the following:

- Risk management and amelioration underlie much of the Rapid Development approach. How do we specifically address metrics comparisons which may be significantly impacted by a Rapid Development methods?
- The structure of Rapid Development methods introduces multiple cycles in the design to test phases. The classic metrics should be amended to reflect the impact of a spiral development model.
- Project management metrics for conventional development applications are centered around concepts designed for conventional development. How do we define a Rapid Development analogue which may be predicated on a different fundamental metric set?
- The metrics need to be extended to include hardware-in-the-loop aspects of the life cycle development. For GN&C systems, the issue of hardware-in-the-loop integration and testing generally has significant impact on the project. Some of these impacts include:
  - Hardware purchase - direct cost of purchasing the hardware elements
  - Hardware acquisition - process costs, not actual hardware costs
  - Familiarization - personnel learning new hardware and adapting it to the RDL
    - √ Training
    - √ Installation
    - √ Checkout
    - √ Acceptance
  - Integration - making the hardware work in the software/hardware construct

    - √ Hardware development (emulators, prototypes, device drivers, etc.
- The use of auto-code generators may impact some traditional measurements.

Appendix A: References

A.1 Text Books

W.W. Agresti, *New Paradigms for Software Development*, IEEE Computer Society Press, 1986.

S.J. Andriole, *Rapid Application Prototyping: The Storyboard Approach to User Requirements Analysis*, QED Technical Publishing Group, 1991.

W. Bischofberger & G. Pomberger, *Prototyping Oriented Software Development: Concepts and Tools*, Springer-Verlag, 1992.

B.H. Boar, *Application Prototyping: A Requirements Strategy for the 80's*, John Wiley & Sons, 1984.

J.L. Connell & L. Shafer, *Structured Rapid Prototyping: An Evolutionary Approach to Software Development*, Yourdon Press Computing Series, 1989.

C. Gane, *Rapid System Development: Using Structured Techniques and Relational Technology*, Prentice Hall, 1989.

S. Hekmatpour & D. Ince, *Software Prototyping, Formal Methods and VDM*, Addison-Wesley, 1988.

K.E. Lantz, *The Prototyping Methodology*, Prentice-Hall, 1985.

M. Mullin, *Rapid Prototyping for Object-Oriented Systems*, Addison-Wesley, 1990.

M.F. Smith, *Software Prototyping: Adoption, Practice and Management*, McGraw-Hill Book Co. (UK), 1991.

A.2 World Wide Web Sites

http://www.lehigh.edu/kaz2/public/www-data/rp/rp.html	RAPID PROTOTYPING
http://www.questicn.com/questicn/radnotes.htm	Rapid Prototyping and Evolution
http://www.cranfield.ac.uk/aero/rapid/rapid_prot.html	Rapid Prototyping
http://cadserv.cadlab.vt.edu/bohn/RP.html	The Rapid Prototyping Resource Center
http://www.iao.fhg.de/Library/rp/OVERVIEW-en.html	Virtual Library on Rapid Product Development
http://www.lookup.com/homepages/56694/scsq/scsq1.htm	Southern California Software Quality
http://stsc.hill.af.mil/www/	Software Technology Support Center Home Page

A.3 Articles & Papers

J.A. Airst & E.J. Norgren, “Rapid Prototyping Using the VMEbus in an Open Systems Architecture Real-Time Environment”, *Proceedings The Fourth International Workshop on Rapid System Prototyping: Shortening the Path from Specification to Prototype*, June 28-30, 1993, Research Triangle Park, NC, pp. 89-99.

S. Alexander, J. Koehler, J. Stolzy, & M. Andre, “A Mission Management System Architecture for Cooperating Air Vehicles”; *Proceedings of the IEEE 1994 National Aerospace and Electronics Conference (NAECON)*, May 23-27, 1994, Dayton, OH; pp. 156-163.

J.M. Ball & E.J. Riel, “Strategic Avionics Technology Program: Rapid Prototyping & Integrated Design Application Studies, Final Report”; NAS9-18877, February 1995.

J.M. Ball, D.C. Weed & E.J. Riel, “Strategic Avionics Technology Program: Rapid Prototyping & Integrated Design Application Studies, White Paper, Rapid Development Process”; NAS9-18877, February 1995.

H.J. Bender, “Alternative Life Cycle Models for Spacecraft Mission Operations Software Development”; *Proceedings of the ESA Symposium: Ground Data Systems for Spacecraft Control*, June 26-29, 1990, Darmstadt, Germany; pp. 443-448.

E.H. Bersoff & A.M. Davis, “Impacts of Life Cycle Models on Software Configuration Management”; *Communications of the ACM*, August 1991, vol. 34 no. 8, pp. 104-118.

B.W. Boehm, “A Spiral Model of Software Development and Enhancement”, *Tutorial: Software Engineering Project Management (R.H. Thayer, ed.)*, Computer Society Press of the IEEE, 1988, pp. 128-142.

B.W. Boehm, *Software Engineering Economics*, Prentice-Hall, 1981.

P. Branton, “More Haste, Less Speed”; *Computer Weekly*, September 7, 1995, p. 37.

A.S. Brown, “Rapid Prototyping-Parts Without Tools”; *Aerospace America*, vol. 29, August 1991, pp. 18-23.

C. Burns, “REE-A Requirements Engineering Environment for Analyzing and Validating Software and System Requirements”; *Proceedings of the Fourth International Workshop on Rapid System Prototyping: Shortening the Path from Specification to Prototype*, June 28-30, 1993, Research Triangle Park, NC; pp. 188-193.

C. Comaford, “Don’t Let Haste Plague Your RAD”; *PC Week*, January 15, 1996, vol. 13 no. 2, p. 18.

J. Connell & G. Wenneson, *Software Engineering Guidebook*, NASA-CR-177625.

L. Constantine, “Under Pressure”; *Software Development*, October 1995, vol. 3 no. 10, p. 112.

L. Constantine, “Re: Architecture”; *Software Development*, January 1996, vol. 4 no. 1, p. 88.

A.M. Davis, “Operational Prototyping: A New Development Approach”; *IEEE Software*, vol. 9 no. 5, September 1992; pp. 70-78.

E.L. Duke, R.W. Brumbaugh & J.D. Disbrow, “A Rapid Prototyping Facility for Flight Research in Advanced Systems Concepts", *Computer*, vol. 22 no. 5, May 1989, pp. 61-66.

A. Dunne, "Rapid Application Development"; *Software Development*, March 1995, vol. 3 no. 3, p. 55.

J.J. Ensell, "Rapid Prototyping for ASIC Designs", *Electronic Engineering Times*, June 19, 1995, no. 853, p. 82.

G.E. Fisher, "Rapid System Prototyping in an Open System Environment"; *Proceedings of the Fifth International Workshop on Rapid System Prototyping: Shortening the Path from Specification to Prototype*, June 21-23, 1994, Grenoble, France, pp. 213-219.

B.R. Givens, "Object-Oriented Applications in a Rapid Prototyping Environment"; *Proceedings of the IEEE 1994 National Aerospace and Electronics Conference (NAECON)*, May 23-27, 1994, Dayton, OH; pp. 814-819 vol. 2.

B.R. Givens & T.R. Hoerig, "A Scalable Architecture for the Rapid Prototyping of Aircraft Cockpits", *Proceedings of the IEEE 1993 National Aerospace and Electronics Conference, NAECON 1993*, May 24-28, 1993, Dayton, OH; vol. 1 pp. 523-528.

I. Glickstein & P. Stiles, "Application of AI Technology to Time-Critical Functions"; AIAA/IEEE Digital Avionics Systems Conference, October 17-20, 1988, San Jose, CA

E.M. Halbfinger & B.D. Smith, "The Range Scheduling Aid"; *Proceedings of the NASA JSC Fourth Annual Workshop on Space Operations Applications and Research (SOAR 90)*, Houston, TX; pp. 280-284.

M. Hanna, "Farewell to Waterfalls"; *Software Magazine*, May 1995, vol. 15, no. 5, p. 38.

J. Harper, "RAD Roundup: A Down and Dirty Tour of the Fast-Paced World of Rapid Application Development"; *HP Professional*, May 1995, vol. 9 no. 5, p. 30.

H.-J. Herpel, M. Held, & M. Glesner; "MCEMS Toolbox: A Hardware-in-the-Loop Simulations Environment for Mechatronic Systems"; *MASCOTS '94, Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, January 31- February 2, 1994, Durham, NC; pp. 356-357.

H.-J. Herpel, N. Wehn, & M. Glesner, "RAMSES-A Rapid Prototyping Environment for Embedded Control Applications", *Proceedings of the Second International Workshop on Rapid System Prototyping: Shortening the Path from Specification to Prototype*, June 11-13, 1991, Research Triangle Park, NC; pp. 27-33.

K. Jones & T. Shepard, "Focusing Software Requirements Through Rapid Prototyping"; *Proceedings of the 1994 Canadian Conference on Electrical and Computer Engineering,* September 25-28, 1994, Halifax, NS, Canada; pp. 629-632 vol. 2.

R.E. Jones, "The Advanced Avionics System Development Laboratory"; *Proceedings of the IEEE/AIAA 10th Digital Avionics Systems Conference*, October 14-17, 1991, Los Angeles, CA; pp. 242-247.

A.F.U. Khan, "Mapping the Expert Systems Development Process to the Department of Defense Life-Cycle Model"; *Proceedings, IEEE Conference on Managing Expert System*

Programs and Projects, September 10-12, 1990, Bethesda, MD; pp. 31-40.

J. King, “Consult Users Early, Often: Price Waterhouse System Goes Up in Just Four Months”; *Computerworld*, October 16, 1995, vol. 29 no. 42, p. 77.

E.M. Lancaster & D.A. Petri, “Program Life Cycle and the System Engineering Process”; JSC-49037, November 1993.

R.-J. Lea, S. Chen, & C.-G. Chung, “On Generating Test Data From Prototypes”; *Proceedings of the Fifteenth Annual International Computer Software and Applications Conference*, September 11-13, 1991, Tokyo, Japan; pp. 345-350.

B.M. Leiner & J.R. Weiss, “Telescience Testbedding: An Implementation Approach”; Research Institute for Advanced Computer Science, NASA Ames Research Center, NASA-CR-185429, 1988.

D.S. Linthicum, “The End of Programming”; *Byte*, August 1995, vol. 20 no. 8, p. 69.

D.S. Linthicum, “RAD: Fast Food for Application Hunger”; *Open Computing*, December 1995, vol. 12 no. 12, p. 48.

M. Lubars, C. Potts, & C. Richter, “A Review of the State of the Practice in Requirements Modeling”, *Proceedings of IEEE International Symposium on Requirements Engineering,* Jan. 4-6, 1993, San Diego, CA; pp. 2-14.

M.D. Lubars, “Reusing Designs for Rapid Application Development”; *ICC 91 International Conference on Communications Conference Record*, June 23-26, 1991, Denver, CO; vol. 3 pp. 1515-1519.

L. Luqi & R. Steigerwald, “Rapid Software Prototyping”; *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, January 7-10, 1992, Hauai, HI; vol. 2 pp. 470-479.

Luqi, “Research Aspects of Rapid Prototyping”; U.S. Naval Postgraduate School, report number NPS52-87-006, 1987.

D. Lyons, “Choosing the Right Tool”; *InfoWorld*, May 1, 1995, vol. 17 no. 18, p. 49.

R.P. Meyer, R.J. Landy & D.J. Halski, “ICAAS Piloted Simulation Evaluation”

M.J. Miedlar & W. Koenig, “Automatic Code Generation for Aerodynamic/Math Models”; *Proceedings of the IEEE 1994 National Aerospace and Electronics Conference (NAECON)*, May 23-27, 1994, Dayton, OH; vol. 2 pp. 839-844.

D. Millington & J. Stapleton, “Developing a RAD Standard”, *IEEE Software*, vol. 12 no. 5, September 1995; pp. 54-55.

H. Mirab & F. Tubb, “First Flight Vehicle Controlled by Computer Generated Software”; *Dynamics and Control of Structures in Space: Proceedings of the 2nd International Conference*, September 6-10, 1993, Cranfield, UK; pp. 653-664.

H. Mirb, “Rapid Prototyping for Real-Time Systems Design”; *Proceedings of the IEE Colloquium on High Accuracy Platform Control in Space*, June 14, 1993, London, UK; pp. 13/

1-5.

C.R. Moffitt, II & Luqi, "A Language Translator for a Computer Aided Rapid Prototyping System"; U.S. Naval Postgraduate School, report number NPS52-88-021, 1988.

J. Mullin, "Rapid Iterative Systems Engineering (RISE)"; *Proceedings of the IEEE Colloquium on Aspects of Systems Integration*, November 12, 1993, London, UK; pp. 4/1-4.

N.C. Olsen, "Designing a Real-Time Platform for Rapid Development"; *Proceedings of the IEEE Workshop on Real-Time Applications*, July 21-22, 1994, Washington, DC; pp. 70-75.

R. Olson, "System Architecture for Process Integration"; *ICSI '92 Proceedings of the Second International Conference on Systems Integration*, June 15-18, 1992, Morristown, NJ; pp. 240-246.

D.F. Oresky & C.W. Haapala, "Verification and Validation in an Iterative Software Development Environment"; *Proceedings of the Fourth International Workshop on Rapid System Prototyping: Shortening the Path from Specification to Prototype*, June 28-30, 1993, Research Triangle Park, NC; pp. 57-67.

M.B. Ozcan & J. Siddiqi, "A Rapid Prototyping Environment Based on Executable Specifications"; *Proceedings of 1994 IEEE Region 10's Ninth Annual International Conference Theme: Frontiers of Computer Technology*, August 22-26, 1994, Singapore; vol. 2 pp. 790-795.

M.B. Ozcan & J.I.A. Siddiqi, "Validating and Evolving Software Requirements in a Systematic Framework", *Proceedings of the First International Conference on Requirements Engineering*, April 18-22, 1994, Colorado Springs, CO; pp. 202-205.

C. Palmer, D. Burwen, E. Finnie, "RASSP Cuts Development Time and Cost"; *Electronic Engineering Times*, July 17, 1995, no. 857, p. 48.

B.K. Patel, V. Litchfield, D. Tamanaha, & A. Davis, "Real Time Systems/Software Methodologies for Large Aerospace Systems", *1991 IEEE Aerospace Applications Conference Digest*, Februrary 3-8, 1991, Crested Butte, CO; pp. 2/1-9.

M.C. Paulk, C.V. Weber, S.M. Garcia, M. Chrissis, & M. Bush, *Key Practices of the Capability Maturity Model, Version 1.1*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania; February 1993.

D. Pesek, "Rapid Development Lab Configuration Management Plan"; internal document, October 1995.

A. Radding, "Managing RAD Programmers: Conventional Management Techniques Often Don't Work"; *InfoWorld*, October 23, 1995, vol. 17 no. 43, p. 74.

S. Rahmani, A.G. Stone, W.S. Luk, & S.M. Sweet, "Rapid Prototyping via Automatic Software Code Generation from Formal Specifications: A Case Study"; *1992 IEEE Aerospace Applications Conference Digest*, February 2-7, 1992, Snowmass, CO; pp. 95-105.

S. Rahmani, "A Software-First Methodology for Definition and Evaluation of Advanced Avionics Architectures", *Proceedings IEEE/AIAA/NASA 9th Digital Avionics Systems*

Conference, October 15-18, 1990, Virginia Beach, VA, pp. 283-288.

Rapid Prototyping Simulation Development Team, “Soyuz ACRV Simulation Development Progress Report, April-May 1993”; McDonnell Douglas TM-6.23.07-24; June 30, 1993.

Rapid Prototyping Simulation Development Team, “Soyuz ACRV Simulation Development Lessons Learned Report”; McDonnell Douglas TM-0009-01 enclosure 1; January 28, 1994.

Rapid Prototyping Simulation Development Team, “Soyuz ACRV Simulation Development Assurance and Test Report”; McDonnell Douglas TM-0009-01 enclosure 2; January 28, 1994

Rapid Prototyping Simulation Development Team, “Soyuz ACRV Simulation Development Configuration Management Plan”; McDonnell Douglas TM-0009-01 enclosure 3; January 28, 1994

Rapid Prototyping Simulation Development Team, “Soyuz ACRV Simulation Development Trip Report, Summary of Trip to MDA-Huntington Beach”; McDonnell Douglas TM-0009-01 enclosure 4; January 28, 1994

J.P. Reilly & E. Carmel, “Does RAD Live Up To the Hype?”; *IEEE Software*, vol. 12 no. 5, September 1995; pp. 24-26.

K. Rizman, I. Rozman, & D. Verber, “The CASE Tool RPT Supporting the Rapid Prototyping Approach to Software Development”, *6th Mediterranean Electrotechnical Conference Proceedings*, May 22-24, 1991, LJubljana, Slovenia, vol. 2 pp. 1033-1036.

K. Rizman, I. Rozman, & D. Verber, “RPT: A CASE Environment Supporting the Rapid Prototyping Approach to Software Development”; *Proceedings of the Advanced Computer Technology, Reliable Systems and Applications 5th Annual European Computer Conference (CompEuro) ‘91*, May 13-16, 1991, Bologna, Italy; pp. 208-212.

R.B. Rowen, “Software Project Management Under Incomplete and Ambiguous Specifications”; *IEEE Transactions on Engineering Management*, vol. 37 no. 1, February 1990; pp. 10-21.

W.W. Royce, “Managing the Development of Large Software Systems”; *Proceedings of IEEE WESCON, 1990*, pp. 1-9.

M. Schrage, “Facilitation: Powerful Job with a Wimpy Name”; *Computerworld*, February 12, 1996, vol. 30 no. 7, p. 35.

B.J. Schroer, F.T. Tseng, S.X. Zhang, W.S. Dwan, *Automatic Programming of Simulation Models, Task 3 Final Report*; NASA-CR-184221, 1990.

J.H. Shackelford, J.D. Saugen, M.J. Wurst, & J. Adler, “The Development of an Autonomous Rendezvous and Docking Simulation Using Rapid Integration and Prototyping Technology” (abstract only); *NASA Automated Rendezvous and Capture Review*, 1993.

J.M. Smith, “In This Game, the Hottest Tools are Totally RAD”; *Government Computer News*, March 20, 1995, vol. 14 no. 6, p. 76.

C. Smyrniotis, “Rapid Prototyping: A Cure for Software Crisis”; *Proceedings of the Twenty-Third Annual Hawaii International Conference on System Sciences*, January 2-5, 1990, Kailua-Kona, HI; pp. 202-210, vol. 2.

W.H. Spuck, “The Rapid Development Method”, Jet Propulsion Laboratory, California Institute of Technology; July 1992.

M.B. Srivastava, T.I. Blumenau, & R.W. Brodersen, “Design and Implementation of a Robot Control System Using a Unified Hardware-Software Rapid-Prototyping Framework”; *Proceedings of the IEEE 1992 International Conference on Computer Design: VLSI in Computers and Processors (ICCD ‘92)*, October 11-14, 1992, Cambridge, MA; pp. 124-127.

A. Stone, C. Scott, & S. Rahmani, “On Definition and Use of Systems Engineering Processes, Methods and Tools”, *1993 IEEE Aerospace Applications Conference Digest*, Jan. 31-Feb. 5, 1993, Steamboat, CO, pp. 197-205.

D.Y. Tamanaha, “Structured Process Flows (SPFs): A Process Model for Metrics”, *1991 IEEE Aerospace Applications Conference Digest*, February 3-8, 1991, Crested Butte, CO, pp. 3/1-12.

D.Y. Tamanaha & P.J. Bourgeois, “Rapid Prototyping of Large Command, Control, Communications and Intelligence C^3I Systems”; *1990 IEEE Aerospace Applications Conference Digest*, February 4-9, 1990, Vail, CO; pp. 253-263.

J.D. Thornton, “Engineering Work Station for Rapid Proto-typing of Real Time Display Systems”; *Proceedings of the IEEE/AIAA 11th Digital Avionics Systems Conference*, October 5-8, 1992, Seattle, WA; pp. 57-61.

C. Tristram, “People Power”; *PC Week*, January 15, 1996, vol. 13 no. 2, p. 13.

J. Uhde & D. Weed, “Library Reuse in a Rapid Development Environment”; *Proceedings of the AIAA Conference on Computing & Aerospace X*, March 28-30, 1995, San Antonio, TX; pp. 521-530.

J. Uhde-Lacovara, D. Weed, B. McCleary, & R. Wood, “The Rapid Development Process Applied to Soyuz Simulation Production”, internal document, 1994

N. Wehn, H.-J. Herpel, T. Hollstein, P. Poechmueller, & M. Glesner, “High-Level Synthesis in a Rapid-Prototype Environment for Mechatronic Systems”, *EURO-DAC ‘92, European Design Automation Conference*, September 7-10, 1992, Hamburg, Germany, pp. 188-193.

R. Weston, “Pushing the Limits of RAD”; *Open Computing*, June 1995, vol. 12 no. 6, p. 30.

Appendix B: Rapid Development Glossary

B.1 Rapid Development Lexicon

The following is a list of terms and their definitions which may be found with some regularity in the literature addressing various topics of a Rapid Development paradigm. There are four sources from which most of the definitions for the terms are derived

- JSC Engineering Directorate; Aerosciences and Flight Mechanics Division; Guidance, Navigation, and Control Rapid Development Laboratory database and experience
- JPL Technical, Commercial and Industrial database and experience
- Technical, Commercial, and Industrial RDM database and experience in the open literature
- IEEE Software Engineering Standards Collection (1994 Edition)

Emphasized text in the definition field of the Glossary indicates that a definition may be found for the emphasized text in the Glossary

TBD signifies the entry is a placeholder to be expanded in a later delivery

AC-100	Hardware in the loop simulation processor (tests **AutoCode** output)
ASDS	Advanced Simulation Development System - Generic trajectory generation and **GN&C/P** simulation tool developed by McDonnell Douglas featuring large reusable libraries of engineering models, utilities, and processes in the Ada language
AutoCode	**MATRIX$_X$** tool for automated translation of **SystemBuild** block diagrams into Ada or C code
Build	Complete integrated and tested, configuration controlled version of system - successive builds.
CASE	Computer Aided Software Engineering - The use of computer based tools to aid in the software engineering process including software design, requirements tracing, code production, testing, document generation, etc.
CI	Configuration Item - An aggregation of hardware and/or software that is designated for **configuration management** (CM) and treated as a single entity in the configuration management process
CM	Configuration Management - A discipline applying technical and administrative direction to identify and document the functional and physical characteristics of a configuration item, control changes to those characteristics, record and report change processing and implementation status, and verify compliance with specified requirements
COCOMO	Constructive Cost Model - Software cost estimation model based on a large database of commercial, industrial, and military software

	applications derived from the book “Software Engineering Economics” by Barry Boehm
concurrent	The occurrence of two or more activities within the same interval of time achieved by either interleaving the activities or by their simultaneous execution
COTS	Commercial off-the-shelf - purchased software tools usually used in the **SEE**
CSC	Computer Software Component - A decomposition of **CSCI**s. May be composed of other **CSC**s or **CSU**s
CSCI	Computer Software Configuration Item - An aggregation of software components that satisfy some end-user function
CSU	Computer Software Unit - The lowest level of **CSCI** decomposition
Deslevs	**Prun** attributes from as developed detailed design
DID	Data Item Description - Essentially a deliverable document
DocumentIt	**MATRIX$_X$** automated documentation and debugging tool providing support for Framemaker, Interleaf, and standard ASCII environments
DOD	Department of Defense
Domain Experts	In the **Rapid Development Model** domain experts provide technical expertise across the required range of technical disciplines in the project. They are members of the **Rapid Development Team** and perform the detailed development functions
Dropspecs	Drop specifications in **Spiral Development Model**
Evolutionary Dev Model	The evolutionary development model features the same strategic basis as the **incremental development model** but differs from it in acknowledging that the user need is not fully understood and all requirements cannot be defined up front. The user needs and system requirements are thus only partially defined up front and refined in each succeeding **build**
Evolution Cycle	A **Rapid Development life** cycle phase resulting in increasingly mature **builds** and leading to the final product at the last cycle
Exit Conditions	Requirement conditions to be met at each milestone
FCA	Functional Configuration Audit - An audit conducted to verify that a configuration item has been completed satisfactorily
FQR	Formal Qualification Review - The test, inspection, or analytical process by which a group of configuration items comprising a system is verified to have met specific contractual performance requirements

GN&C	Guidance, Navigation, and Control
GN&C/P	The **GN&C** system combined with the propulsion system
GUI	Graphical User Interface - Generic term for utilization of screen data presentation and user input via pointing devices (e.g. mouse) to facilitate user interaction with a software construct
HCE	Hardware Connection Editor HCE
HWCI	Hardware Configuration Item - An aggregation of hardware components that satisfy some end-user function
Incremental Dev Model	A software development technique in which the requirements, design, implementation and testing occur in an overlapping, iterative, manner resulting in incremental completion of the overall product
IDD	Interface Design Description - A document defining interfaces between **CSCI**s
ISO 9000	International Organization for Standards method for assessing supplier ability to meet commitments and requirements (International analogue of **SEI CMM**)
IV&V	Independent Verification and Validation
KPA	Key Process Area - **CMM** area of focus
MATRIXx	An integrated toolset providing a graphical environment for analysis and development of system requirements, design, development, code, and test over the entire development cycle
Metric	Quantitative measure of system size, complexity, cost, quality etc.
OCR	Operational Concept Review -.Reviews held to resolve open issues regarding the operations concept for a system
PCA	Physical Configuration Audit - An audit conducted to verify that an as built configuration item conforms to the technical documentation that defines it
PMI	Project Management Information - the Information required by each **Prun** in addition to the requirements to be used in project management and design
Pruns	Projects Units (HW & SW packages, **Superblocks**, models, etc.)
Prototyping	A hardware and software development technique in which a preliminary version of the hardware/software product is developed to stimulate user feedback, determine feasibility, or investigate timing or other issues in support of the development process
QA	Quality Assurance - TBD
Quality Gates	The set of conditions which must be met to transition from one life

	cycle phase to the succeeding phase
Rapid Dev Model	A extension of the **Spiral Development Process** where additional tools (such as integrated **SEE** tools like **MATRIXx** are used to speed the process
Rapid Development Team	In the **Rapid Development** paradigm a project dedicated team which assumes ownership of the entire development process and end product. The Rapid Development Team includes all critical domain and systems skills and expertise needed to successfully complete the project.
Rapid Prototyping	A subset of the **Rapid Development** process where an initial prototype version is created, primarily for validating the initial requirements and design concepts
RC	Requirements Change - The reversal of the conventional acronym CR (Change Request) is intended to specify the evolutionary requirements discovered during the Rapid Development evolutionary model exercise as opposed to a conventional new requirement written against the requirements baseline
RDL	Rapid Development Laboratory - NASA/JSC facility for accelerated **GN&C** software development research and applications
Reqlevs	**Prun** attributes from high level requirements (requirements “shalls”)
Reusable Library	Collection of reusable code modules (e.g. utilities, models, etc.)
SDF	Software Development File - A collection of material pertinent to the development of a given software unit or set of related units. Contents typically include the requirements, design, technical reports, code listings, test plans, test results, problem reports, schedules, and notes for the units
SEE	Software Engineering Environment - The hardware, software, firmware, procedures and documentation needed to perform software engineering. Elements may include but are not limited to **CASE** tools, compilers, assemblers, linkers, loaders, operating systems, debuggers, simulators, emulators, documentation tools, and database management systems
SEI	Software Engineering Institute
SLOC	Source lines of code - **metric** for sizing of software products
Spiral Dev Model	Spiral Development Process - An accelerated development process where the system requirements, design, code, test, and integrated test processes are iterated on concurrently rather than being executed sequentially
Statemate	Methods and tools for requirements, development, and validation

STE	Software Test Environment - The facilities, hardware, software, firmware, procedures and documentation needed to perform qualification, and possibly other, testing of software. Elements may include but are not limited to simulators, code analyzers, test case generators, path analyzers, etc. and may include elements used in the **SEE**
Superblocks	TBD
Support Processes	The set of general technical, management, and institutional processes providing support to the development process (e.g. project management, requirements management, configuration management, etc.)
SysemBuild	**MATRIX**$_{\mathbf{X}}$ graphical interface tool supporting system design from data flow block diagrams
Test Case Spec.	A document that specifies the test inputs, execution conditions and predicted results for an item to be tested
Test Design	Documentation specifying the details of a test approach for a software feature or combination of features and identifying the associated tests
V&V	Verification and Validation - Determination whether the requirements for a system or component are complete and correct, the products of each development phase fulfill the requirements and conditions imposed by the previous phase, and the final system or component complies with specified requirements
Waterfall Model	Conventional software development model featuring sequential development of the component life cycle phases with requirements established early in the process
Xmath	**MATRIX**$_{\mathbf{X}}$ tool for design and analysis of simulations, control systems, and numerical calculations

B.2 MIL-STD-498 Reviews and Documentation

The following is a list of terms associated with documentation and technical/management reviews suggested in MIL-STD-498. The MIL-STD-498 is superseding MIL-STD-2167A and makes many significant changes to accommodate currently evolving Rapid Development processes

The source from which the terms for this Appendix are derived is:

- MIL-STD-498 - Software Development and Documentation, Software Technology Support Center, Hill Air Force Base, Utah,

Emphasized text in the definition field of an entry indicates that a definition may be found for the emphasized text in the elsewhere in the Appendix

TBD signifies the entry is a placeholder to be expanded in a later delivery

498	MIL-STD-498 - **DOD** Standard superseding MIL-STD-**2167A** for the development of systems and software applications. A key feature of the standard is its strategy for accommodating facets of the **Rapid Development model** paradigm
1521B	MIL-STD-1521B - **DOD** Standard for technical reviews and audits invoked by **2167A**
2167A	MIL-STD-2167A - **DOD** Software Development Standard - Establishes uniform requirements for software development applicable throughout the system life. The standard provides the basis for Government insight into contractor software development, testing, and evaluation
CDM	Conventional Development Model - Software development paradigm predicated on a sequential development model of the software/ hardware system. The CDM underlies the structure of the system life cycle in **2167A**
CDR	Critical Design Review - Formal review required by **1521B** to review the detailed designs for each **CSU** and assure the system configuration items meet the specified requirements
CMM	Capability Maturity Model - **SEI** method for assessing supplier ability to meet commitments and requirements (American analogue of **ISO 9000**)
COM	Computer Operation Manual - A **498 DID** containing instructions for operating a computer
CPM	Computer Programming Manual - A **498 DID** containing instructions for programming a computer
CRR	Critical Requirements Review - Reviews held to resolve open issues regarding the handling of critical requirements, such as safety, security, and privacy

DBDD	Database Design Documentation - A **498 DID** describing the design of an associated database
FSM	Firmware Support Manual - A **498 DID** containing instructions for programming firmware devices
IDD	Interface Design Document - A **498 DID** specifying the design of one or more interfaces between one or more software systems and hardware systems
IPR	In-Progress Review - Technical and management reviews scheduled at completion of a **Build** milestone
IRS	Interface Requirements Specification - A **498 DID** containing the requirements for one or more interfaces
OCD	Operational Concept Description - A **498 DID** containing the operational concept for the system
PDR	Preliminary Design Review - Formal review required by **1521B** to review the detailed designs of **CSCI**s and **CSC**s to evaluate the progress, technical adequacy, and risk resolution of the selected design approach for one or more configuration items
SCOM	Software Center Operator Manual - A **498 DID** containing instructions for operators of a batch or interactive software system that is installed in a computer center
SDD	Software Design Description - A **498 DID** containing detailed design for each **CSCI**
SDP	Software Development Plan - A **498 DID** containing the plan for performing the software development activities
SDR	Software Design Review - Reviews held to resolve open issues regarding the architectural design of a **CSCI**, **CSCI**-wide design decisions, and detailed design of a **CSCI** or portion thereof (such as a database)
SIOM	Software Input/Output Manual - A **498 DID** containing instructions for users of a batch or interactive software system installed in a computer center
SIP	Software Installation Plan - A **498 DID** containing the plan for installing the software at the user sites
SPR	Software Plan Review - Reviews held to resolve open issues regarding the **SDP,** STP, SIP, and STrP
SPS	Software Product Specification - A **498 DID** comprised of the executable software, the source files, and information to be used for support
SQPP	Software Quality Program Plan - A description of the plan for

	implementing quality control procedures in a hardware/software system development
SwRR	Software Requirements Review - A review of the requirements specified for one or more software configuration items to evaluate their responsiveness the system requirements
SyRR	System Requirements Review - A review of the completeness and adequacy of the system requirements to evaluate the system engineering process that produced those requirements and to assess the results of system engineering studies
SSDR	System/Subsystem Design Review - Reviews held to resolve open issues regarding the system or subsystems design decisions or the architectural design
SSS	System/Subsystem Specification - A **498 DID** providing documentation of the essential requirements (functions, performance, design constraints, and attributes) of the software system
SRR	Software Requirements Review - Reviews held to resolve open issues regarding the specified requirements for a **CSCI**
SRS	Software Requirements Specification - A **498 DID** presenting the requirements to be met by each **CSCI**
SSDD	System/Subsystem Description - A **498 DID** defining the system and its. partitioning into **HWCI**s and **CSCI**s
SSR	Software Supportability Review - Reviews held to resolve open issues regarding the readiness of the software for transition to the support agency, the software product specifications, the software support manuals, the software version descriptions, and the status of transition preparation and activities
SSRR	System/Subsystem Requirements Review - Reviews held to resolve open issues regarding the specified requirements for a software system or subsystem
STD	Software Test Description - A **498 DID** containing test case descriptions and procedures for qualification testing for one or more software systems
STP	Software Test Plan - A **498 DID** containing the plan for conducting qualification testing
STR	Software Test Report - A **498 DID** containing a record of the formal qualification testing performed on the software system
STrP	Software Transition Plan - A **498 DID** containing the plan for transitioning to the support agency

SUM	Software Users Manual - A **498 DID** containing instructions sufficient to execute a software system
SUR	Software Usability Review - Reviews held to resolve open issues regarding the readiness of the software for installation at user sites, user and operator manuals, software version descriptions, and the status of installation preparation activities
TRdR	Test Readiness Review - Reviews held to resolve open issues regarding the status of the **software test environment,** the status of the software to be tested, and the test cases and procedures to be used for **CSCI** qualification testing or system qualification testing
TRsR	Test Results Review - Reviews held to resolve open issues regarding the results of **CSCI** testing or system qualification testing
SVD	Software Version Description - A **498 DID** which identifies and describes a version of a system or component

Appendix C: Metrics Glossary

The following is a list of terms and their definitions which are associated with the application of metrics to the Software/Hardware development and sustaining engineering processes. There are three sources from which most of the definitions for the terms are derived

- DA3 Software Development Metrics Handbook Version 2.1 - NASA/JSC 25519 April 1992
- DA3 Software Sustaining Engineering Metrics Handbook Version 2.0 - NASA/JSC 26010 December 1992
- DA3 Development Project Metrics Handbook Version 5.0 - NASA/JSC 36112 March 1993

The following Metrics Glossary items do not include project management metrics.

Active Test Time	Elapsed wall-clock time during which software was actively being tested. Active test time does not include time lost due to failure, reconfiguration, or debugging
Actual SLOC	A count of the lines of New and Modified code actually produced during the code production and test phases of a project.
Baseline SLOC	The size of the current operational baseline at project start. Applies to a project to update or modify an existing program.
Break/Fix Ratio	The number of DRs closed with a software fix that were generated as the result of a previous DR fix or CR enhancement divided by the sum of the number of DRs closed with a software fix plus the number CRs closed with a software change
Capacity	Maximum amount of a resource available for use.
Change Request (CR)	A request for system enhancement
Closed Date	The date when the software is returned to operations or is operationally ready. (i.e. the software fix or enhancement is complete and on the floor).
Closure Codes	A classification scheme used to identify how work was completed or submitted on DR requests
Code and Test	Code is the translation of a designed unit into a computer program that can be accepted by a processor. Testing is the exercising (either manually in the case of a walkthrough or electronically through unit test cases or both) of the coded unit
Comment Ratio	The fraction of the number of comment lines in the new and modified software to the SLOC in the new and modified software.
Comment SLOC	A textual string, line, or statement that has no effect on compiler or program operations
Corrective Maintenance	Software changes resulting from Discrepancy Reports

COTS	Commercial-off-the-shelf software that is purchased for a project. COTS software is only included in the software size metric if it is to be maintained by the purchaser
Compilation Unit	The lowest independently compilable software element subject to configuration management.
CSC	Computer Software Component - equivalent to an Ada software package
Computer Resource Utilization	The fraction of time a resource is busy.
Configuration Item	Logically related grouping of units or packages which perform a major function of the system.
Critical DR	A failure that affects the following systems in the manner described: Development System - Inhibits major processing in more than one area and cannot be circumvented. Test System - Inhibits one or more applications from being tested, or brings the system to a halt and cannot be circumvented. Operational Systems - Drastically reduces the usefulness of the system in support of current operations, and cannot be circumvented. All Systems - Requires reboot of workstation to correct problem.
Data Primitive	A basic data item required to compute a metric value.
Defect	An error in the software.
Deleted SLOC	Existing SLOC that will be removed from the baseline by the completion of the delivery.
Design	The definition of each software unit’s control and data structure, interfaces, and lists of accessed data items.
Development Progress	A measure of progress toward design, implementation, and integration of the software.
Discrepancy Report	A notification that a system under test or in operation (i.e. hardware, software, system, operations) has deviated from the behavioral characteristics expected of it. The notification carries a description of the problem, an assessment of the criticality of the problem, and a portion of the system to which the problem is charged.
DR Criticality	The assessed effect the DR has on the continuance of the system activity.(e.g. test, mission support, training. etc.)
DR Density	The total number of DRs written against a piece of software divided by the size of that software.
Error	A human action taken during the design, code, or test of software that results in a fault.
Fault Type	A problem identifier which may be categorized by the closure code of

	the DR, by the DR criticality, or by the taxonomy of faults established for the project.
Failure	The inability of a system or system component to perform a required function within specified limits.
Failure Rate	The cumulative number of failures divided by the cumulative active test time.
Fault Density	The total number of DRs written against a piece of software divided by the size of that software.
Generic Unit	A template that defines a program unit as either a generic subprogram or a generic package.
Integration	The process of combining coded and tested units and configuration items into a system or subsystem.
Instance	Specified subprograms or packages that are obtained by assigning values to the parameters of the generic unit.
Major DR	A failure that affects the following systems in the manner described: Development System - Inhibits major processing or produces erroneous outputs limited to one function Test System - Inhibits an entire processor of an application from being tested or prohibits completion of a test case by blocking other test functions. Operational System - Reduces the usefulness of one or more major system functions used in the current operations, and cannot conveniently be circumvented. All Systems - Logoff/Logon is required to restore operation.
Minor DR	A failure that affects the following systems in the manner described: Development System - Anomalies that slight and can be circumvented Test System - DRs that do not directly affect completion of a test function and are considered to have no effect or to be insignificant in an operations environment. Operational System - DRs that occur during a mission, simulation, or validation period that are considered to have no effect or to be insignificant during that period.
McCabe Complexity	The number of linearly independent paths in a module that, when taken in combination, will generate every possible path.
Modified SLOC	A module or compilation unit is “modified” if it is changed and the change affects less than 20% - 50% of the SLOC.
Module	The lowest level of software compilation subject to configuration management.
New SLOC	Newly developed code or code in a module or compilation unit that has been changed and the changes affect more than 20% - 50% of its

	SLOC.
Normalized Active Test Time	Active test time divided by the sum of new plus modified SLOC.
Observed Failure Rate	The cumulative number of failures divided by the active test time.
Open DR Density	The total number of open DRs written against a piece of software divided by the size of that software.
Operational Hour	An hour that the system is directly supporting a primary user.
Progress	A count of the number of DR or CR requests submitted minus the number of DR or CR requests closed during a specified reporting interval.
Project	The set of activities performed to develop a new system or to upgrade an existing system.
Release	The entire software configuration, not just the changed modules.
Resource	An available system component.
Reused SLOC	Those SLOC that are not part of the baseline and exist on a different project but are used on the current project.
Software Reliability	The probability that the software will not cause a failure of a system for a specified time under specified conditions.
Software Requirement	Any “shall” statement in the project’s controlling software specification.
Software Staff	All those directly involved in the software development activity, including programmers, testers, and first line managers.
SLOC	An acronym for Source-Line-Of- Code, any non-comment, non-blank carriage return terminated source line of code.
Staffing	The number of hours spent on a project by all those directly involved in the development activity, including programmers, testers, and first line management.
Subsystem	A collection of functionally related software configuration items.
System	A group of interacting, interrelated, or interdependent elements (sub-system or other configuration components) which form a recognized complex whole.
Subprogram	A sequenced set of statements that may be used in one or more computer programs and at one or more points in a computer program.
Test Baseline SLOC	Those unmodified baseline SLOC that must be retested to verify system operational requirements are met.
Test Case	A specific set of procedures using associated data developed to

	exercise a particular program path or verify compliance with a specific requirement.
Total SLOC	Total SLOC is defined as (Unmodified SLOC + New SLOC + Modified SLOC - Deleted SLOC + Reused SLOC)
Unit	A collection of modules or compilation units performing a testable function
Unmodified Baseline	The number of current operational baseline SLOC that are not changed during the development effort.
Workload	The collective designation of a system’s inputs (i.e. programs, data, commands) over time.

Appendix D: Traditional Metrics Definitions and Acronyms

One effective metrics program in use at NASA, in a traditional development environment, was adopted by the JSC/Mission Operations Directorate (MOD) during the period from May 1990 to March 1992. This metrics program has been and is being used successfully as a management tool for large safety critical projects, and provide a good starting point for developing a metrics program. A summary of the metrics collected is presented here for reference.

The source documents which describe the program in detail include:

Software Development Metrics Handbook, JSC-25519, Version 2.1, April 1992

Software Sustaining Engineering Metrics Handbook, JSC-26010, Version 2.0, December 1992

Development Project Metrics Handbook, JSC-36112, Version 5.0, March 1993

D.1 Definitions

Three tables of metrics definitions are presented here. These represent samples of traditional Development Engineering, Sustaining Engineering, and Project Management metrics. Typically, a new metrics program can be expected to implement on Development Engineering metrics, with the others added progressively as the program matures. A variety of acronyms which appear in these tables are expanded in section D.2

Table 3. Sample of Traditional Development Engineering Metrics

Metric Classification	Description	Data Collected
Software Size	The SLOC in the system that must be tested and maintained	Total SLOC New SLOC Modified SLOC Reused SLOC Baseline SLOC Deleted SLOC Test baseline SLOC Unmodified SLOC Unmodified Baseline SLOC Ratio of Comments to Total SLOC
Software Staffing	Number of engineering and first line management personnel involved in system development	Planned staff hours Actual staff hours
Software Requirements Stability	Total number of requirements to be implemented for the project	Total A,B,C-level reqts (“shalls”) Cumulative changes

Table 3. Sample of Traditional Development Engineering Metrics

Metric Classification	Description	Data Collected
Development Progress	Number of modules successfully completed from design through test	Planned and actual “units” designed Planned and actual “units” coded Planned and actual “units” tested Planned and actual “units” integrated
Computer Resource Utilization	Percent of CPU, disk, memory, and I/O channel utilization	CPU utilization/capacity disk utilization/capacity memory utilization /capacity I/O channel utilization /capacity
Test Case Completion	Percent of test cases successfully completed	Planned system integration tests Actual system integration tests
Discrepancy Report Open Duration	Time lag from problem report initiation to problem report closure	Critical DRs closed in <10 days Critical DRs closed in <30 days Critical DRs closed in <60 days ..... etc.
Fault Density	Open and total defect density over time	New SLOC Modified SLOC Total DRs written Total DRs closed Active test hours
Test Focus	Percent of problem reports closed with a software fix	Total DRs closed Total DRs closed with a single fix
Software Reliability	Probability that the software works for a specified time under specified conditions	Cumulative critical DRs written Cumulative major DRs written Cumulative minor DRs written Active test hours
Design Complexity	Number of modules with a complexity greater than an established threshold	Number of modules with McCabe > 10 Number of modules with McCabe > 40 ..... etc.
Ada Instantiations	Size and number of generic subprograms developed and the frequency of their use within the project	Number of generic units developed Number of instances of the generic unit Generic unit SLOC count

Table 4. Sample of Traditional Sustaining Engineering Metrics

Metric Classification	Description	Data Collected
Break/Fix Ratio	Ratio of DRs resulting from a DR fix or SR change to the total number DRs+SRs	No. of DRs changed with software fix No. SRs closed with software change DRs closed with a software fix generated with a previous DR fix or SR enhancement
Software Volatility	Number of times a module is changed due to a Service Request	Number of modules changed per release Total modules in release Release date
SR Scheduling	The length of time to close an SR and the effort spent on SR closure	Date of submission Date of availability for release inclusion Date of SR release to facility
Problem/ Enhancement Closure	Actual DR and SR closure rate by (sub)system	DRs submitted/closed by (sub)system SRs submitted/closed by (sub)system
Fault Type Distribution	Percent of defects closed by type of fault (e.g. logic, error handling, standards, interface, etc.)	Number of DRs closed by category Number of DRs closed by code Number of DRs closed by fault type Number of DRs closed by process identity
Staff Utilization	Staff effort for DRs by (sub)system Staff effort for SRs by (sub)system	Effort per DR open/closed by (sub)system Effort per SR open/closed by (sub)system

Table 5. Sample of Traditional Project Management Metrics

Metric Classification	Description	Data Collected
Schedule Performance	Milestone Volatility for the Next Year	MVYI = MAM/TM
	Project Critical Path Performance Index	$PCPI_i = ACPMI_i/SCPM_i$
	Reporting Period Milestone Performance Index	$MPI_i = AM_i/SM_i$
	Cumulative Milestone Performance Index	CMPI = CAM/CSM
	Project Schedule Deviation on Critical Path	PAT = (EBT + 4*MLT + EWT)/6
	Earned Value Rate to Completion Rate Index	EVCRI = AER/MER
	Schedule Performance Index	$SPI_i = BCWP_i/BCWS_i$
	Acquisition Performance Index	$API_i = AAM_i/SAM_i$
Cost Performance	Budget Performance Index	$BPI_i = BU_i/AU_i$
	Cumulative Budget Performance Index	CBPI = CBU/CAU
	Staffing Index	$SI_i = 100*(ASRU_i - PSRU_i)/PSRU_i)$
	Cost Performance Index Using EV	CPI = BCWP/ACWP
Cost-Schedule Performance	Project Performance Index	PPI = (MPI + BPI)/2
	Cumulative Project Performance Index	CPPI = (CMPI + CBPI)/2
	Cost Schedule Index Using EV	CSI = (SPI + CPI)/2

Table 5. Sample of Traditional Project Management Metrics

Metric Classification	Description	Data Collected
Delivered Content	Cumulative Risk Performance Index	CRPI = RRK/TRK
	Reviews and DRLIs Performance Index	$RDPI_i = (AMR_i/PMR_i + ADRL_i/PDRL_i)/2$
	Hardware and Software Delivery Performance Index	$HWSWI = K_1*AHW/PHW + K_2*ASLOC/PSLOC$
	Requirements Volatility from Baseline Index	RVBI = $(\text{Changed} + \text{Added} + \text{Deleted})_{REQTS}$ /Total Baselined Requirements
	Training Performance Index	$TPI_i = ATH_i/PTH_i$
Quality Performance	DRL Rework (RIDs or CRs) Average	$DRLRA_i = (S(RID_{ij})_{j=1,Ni})/N_i$
	Hardware Development DRs Rate	HWDRR = HWDR/HWUT
	Software Development DRs Rate	SWDRR = SWDR/SWIT
	DR Rate	DRR = DR/TR
	Quality Point Reviews Index	QPRI = AQPR/PQPR
	Average Quality Point Review Score	$AQPRSi = (S(QPS)_{j=1,Mi})/M_i$
External Influences	Reqts Volatility from Baseline Impact	Estimates of dollar or time impacts due to: changing requirements; untimely decisions; project dependency items; unscheduled work
	Timeliness of Decision and Proj Dep Items	
	Unscheduled Work Impact	
	Subsystems Impacted by Reqts Deviation	
	Requirements Complexity	

D.2 Metrics Acronyms

AAM	achieved acquisition milestones
ACPM	achieved critical path milestones
ACWP	actual cost of work performed
ADRLI	actual DRLIs
AEVR	actual earned rate
AHW	actual delivered hardware subsystems or units
AM	achieved milestones
AMR	actual major reviews
AQPR	actual quality point review
ASI	actual staff resource units
ASLOC	actual delivered source lines of code
ASRU	actual staff resource unit
AT	acceptance testing
ATH	actual training hours
AU	actual units
BCWP	budgeted cost of work performed
BCWP-Cum	cumulative budgeted cost of work performed
BCWS	budgeted cost of work scheduled
BPI	budget performance index
BU	budgeted units
CAM	cumulative achieved milestones
CASE	Coordination and systems engineering
CAR	cumulative actual risk (Sum of IDR_i
CAU	cumulated actual units
CBPI	cumulative budget performance index
CBU	cumulative budgeted units
CMPI	cumulative milestone performance index
COTR	contracting officer technical representative
COTS	commercial off the shelf
CPM	critical path method
CPI	cost performance index
CPPI	cumulative project performance index
CR	change request
CSI	cost schedule index
CSM	cumulative scheduled milestones
CV	cost variance
DR	discrepancy report
DRD	data requirements description
DRL	data requirements list
DRLIs	data requirements list items

DRLID	data requirements list items delivered (N_i)
DRLRA	data requirements list (DRLs) rework average
DRR	DR rate
EBT	estimated best time
EV	earned value
EVCRI	earned value rate to completion rate index
EWT	estimated worst time
GFE	government furnished equipment
HWDRR	hardware development DRs rate
HWUT	hardware units in test or operating at time of testing
IDR_i	Identified risk per reporting period
K_1	weighted relative importance of hardware
K_2	weighted relative importance of software
KSLOC	thousands of source lines of code
MAM	moved, added, and deleted milestones
MEVR	minimum earned value rate
MPI	milestone performance index
MRK	mitigated risk
MT	mode time
MVYI	milestone volatility for the next year
PAT	projected activity time
PCPI	project critical path performance index
PDRLI	planned DRLIs
PERT	program evaluation and review technique
PHW	planned hardware subsystems or units
PMR	planned major reviews
PPI	project performance index
PQPR	planned quality point review
PRUN	project unit
PSLOC	planned source lines of code
PSI	planned staff resource units
PSR	project status review
PSRU	planned staff resource unit

PTH	planned training hours
QPS	quality point scores
QT	qualification testing
RC	revealed requirements change
RID	review item disposition
RRK	remaining risk
RTP	remaining time period
RVBI	requirements volatility from baseline index
SAM	scheduled acquisition milestones
SCPM	scheduled critical path milestones
SDCP	schedule deviation on the critical path
SDDR	system detailed design review
SEO	system engineering office
SFDR	system functional design review
SM	scheduled milestones
SOP	standard operating procedure
SPI	schedule performance index
SRR	system requirements review
SSAT	subsystem acceptance test
SSCDR	subsystem critical design review
SSIT	software in KSLOC in test
SSPDR	subsystem preliminary design review
SV	schedule variance
SWDR	software DRs
SWDRR	software development DRs rate
TM	total milestones
TR	test run
TRK	total risk